RANK XEROX

RANK XEROX EXTENDED ALGOL-60

Sigma 5-9 Computers

Language and Operations Reference Manual

190572C

MAY 1972

PRICE: £1-50

Rank Xerox- Data Systems Division, York House, Empire Way, Wembley, Middlesex, England.

Printed in England

ACKNOWLEDGMENT

The RXDS EXTENDED ALGOL 60 Compiler was developed by Robert P. Cook and Gilbert J. Hansen under the direction of Dr. George Haynam of Evergreen Associates, Winter Park, Florida, U.S.A.

This Document reflects the COO Release of Algol-60.

RELATED PUBLICATIONS

Title	Publication No.
RXDS Sigma 5 Computer Reference Manual	90 09 59
RXDS Sigma 7 Computer Reference Manual	90 09 50
RXDS Sigma 8 Computer Reference Manual	90 17 49
RXDS Sigma 9 Computer Reference Manual	90 17 33
RXDS Sigma 5/7 Batch Processing Monitor Reference Manual	90 09 54
RXDS Sigma 5/7 Batch Processing Monitor Operations Manual	90 11 98
RXDS Sigma 5/7 FORTRAN IV-H Reference Manual	90 09 66
RXDS Sigma 5/7 FORTRAN IV Reference Manual	90 09 56
RXDS Sigma 5/7 FORTRAN IV Library Technical Manual	90 15 24

Notice

The Specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their RXDS sales representative for details. ACM

TABLE OF CONTENTS

I. INTRODUCTION

1.1 System Requirements..... 1
1.2 Organization of the Manual..... 1
1.3 The Translator..... 2
1.3.1 Deviation from Algol 60..... 2
1.3.2 Basic Symbols..... 4

II. ELEMENTS OF THE COMPILER LANGUAGE

2.1 Character Sets..... 7
2.2 Identifiers..... 8
2.3 Quantities..... 9
2.4 Variables..... 10
2.5 Constants..... 11
2.5.1 Integer Constants..... 11
2.5.1.1 Hexadecimal Constants. 11
2.5.1.2 Bit Constants..... 11
2.5.2 Real Constants..... 12
2.5.3 Real 2 Constants (Double Precision)..... 12
2.5.4 Boolean Constants..... 13
2.5.5 String Constants..... 13
2.5.6 Complex Constants..... 13
2.6 Functions or Evaluated Procedures... 14

III EXPRESSIONS

3.1 Arithmetic Expressions..... 15
3.1.1 The Conditional Arithmetic Expression..... 16
3.1.2 Strings in Arithmetic Expressions..... 16
3.1.3 Construction of Arithmetic Expressions..... 17
3.2 Boolean Expressions..... 18
3.2.1 Relations..... 19
3.2.1.1 Strings in Relations.. 20
3.2.1.2 Complex Quantities in Relations..... 20

3.2.2 The Conditional Boolean Expression..... 20
3.2.3 Construction of Boolean Expression..... 21
3.3 String Expressions..... 21
3.4 Bit Expressions..... 22
3.5 Designational Expressions..... 22
3.6 Precedence of Operators..... 23

IV STATEMENTS

4.1 The Assignment Statement..... 24
4.1.1 Arithmetic Assignment Statements..... 25
4.1.2 String Assignment Statements..... 25
4.1.3 Bit Assignment Statements..... 26
4.1.4 Boolean Assignment Statements. 27
4.1.5 Generalized Assignment Statement..... 27
4.2 The Grammar of Statements..... 28
4.3 Compound Statements..... 28
4.4 Statement Labels..... 29
4.5 DUMP Statement..... 30
4.6 The COMMENT..... 31

V. BASIC DECLARATIONS

5.1 Declarations of Type..... 32
5.2 The STRING Declaration..... 33
5.2.1 Construction of String Declarations..... 33
5.3 The ARRAY Declaration..... 35
5.3.1 Construction of Array Declarations..... 35
5.3.2 The STRING ARRAY Declaration.. 36
5.4 The OWN Declaration..... 37
5.5 The SWITCH Declaration..... 38
5.6 The LOCAL Declaration..... 39
5.7 DEFINE Declaration..... 40
5.7.1 Nesting of Definitions..... 41
5.8 The EXTERNAL Declaration..... 41

VI. CONTROL STATEMENTS

6.1 The GO TO Statement..... 43
6.2 The IF Statement..... 44
6.3 The FOR Statement..... 46
6.3.1 Jumps in and out of FOR Statements..... 52

VII. STANDARD FUNCTIONS

7.1 Intrinsic Functions..... 53
7.2 Type Transfer Functions..... 55
7.3 Library Functions..... 56
7.3.1 Input/Output Procedures..... 56
7.3.2 Standard Mathematical Procedures..... 57
7.3.3 Recursive Procedures..... 58

VIII INPUT/OUTPUT

8.1 Input/Output..... 59
8.1.1 The READ Statement..... 59
8.1.2 The WRITE Statement..... 62
8.1.3 BCD and BINARY Input/Output..... 64
8.2 The FORMAT Declaration..... 66
8.3 The ERR and EOF Statements..... 67
8.4 The REWIND, BACKSPACE and ENDFILE Statements..... 67
8.4.1 REWIND Statement..... 68
8.4.2 BACKSPACE Statement..... 68
8.4.3 ENDFILE Statement..... 68
8.5 STRING Input/Output..... 69

IX. BLOCKS

9.1 Block Format..... 70
9.2 Defining a Block..... 71
9.3 Local and Global Identifiers..... 72

X. PROCEDURES

10.1 Internal Procedures..... 75
10.1.1 The Procedure Block..... 75
10.1.1.1 The PROCEDURE Declaration..... 75
10.1.1.2 The VALUE Part..... 76
10.1.1.3 The Specification Part. 77
10.1.2 Value and Name Parameters..... 80
10.1.3 Functional Procedures..... 80
10.1.4 The Procedure Call..... 82
10.1.5 Copy Rule..... 83
10.1.6 Recursive Procedure Calls..... 84
10.1.7 General Problem Solver..... 85
10.2 External Procedures..... 86
10.2.1 Procedure Declaration..... 87
10.2.2 Procedure Calls..... 87
10.2.3 Operational Features..... 90

XI. OPERATING FEATURES

Processor Call..... 92
Options..... 92
Source Card Format..... 95
Fast Loader..... 95
Sample Forms..... 96

Appendix A XDS-Fortran Format Description
Appendix B BNF for Input/Output
Appendix C Reserved Identifiers
Appendix D Error Messages

PREFACE

This manual is a user's reference manual for the extended Algol 60 compiler on the Sigma 5-9. Every effort has been made to make this compiler adhere as closely as possible to the revised Algol 60 language specifications (Communications of the ACM, Vol. 6, January 1963, 1-17). Desirable extensions have been made to the source language to adapt it for the handling of data processing problems while retaining the conciseness of notation used in scientific work. The extensions include features found in the Burroughs B5000 Algol such as bit manipulation, strings, and DEFINE macros. Strings have been generalized to be a dynamic type which applies to simple variables, subscript variables and procedures. Facilities have been provided for defining simple sub-string variables and sub-string arrays. The facilities provide for string manipulation, and allow the comparison, concatenation, and arithmetic operations between string elements. This generalization provides most of the facilities available in the COBOL language. Finally, external variables, labels, and procedures have been added to the language to permit procedures to be separately compiled.

SYSTEM REQUIREMENTS

ORGANIZATION OF THE MANUAL 1....

THE TRANSLATOR INTRODUCTION

This is intended as a reference manual in the use of an extended Algol 60 Language, based in part on the "Revised Report on the Algorithmic Language on the ALGOL 60" (Communications of the ACM, Vol.6, January 1963, 1-17). Algol language and produces machine-language programs for the XDS Sigma 5-9 Computer.

The Algol translator utilizes a XDS Sigma 5-9 consisting of at least the following components: 48K words of core storage, one card input station, one high speed printer, and at least three megabytes of RAD storage.

The text of this reference manual consists principally of definitions and rules for the use of the translator, examples of these rules,and some sample programs. A set of appendices summarizes the text and lists some details on the operation of the program, the contents of the library etc.

Whenever a term is defined, it is underlined in the defining sentence. Greek letters or names enclosed in corner brackets (e.g. ⟨integer⟩ are used in the text to denote generic representations; for example, ξ is used to represent an expression and Σ to represent a statement. For the most part, other symbols represent themselves.

The exaples, which have been used quite liberally, have been employed for "definitions by example" in only those few cases where a formal description has proved particularly unwieldy.

1.3 THE TRANSLATOR

The Computer's use of the Algol translator is split into two distinct phases (1) first, all of the statements of the program in Algol language are translated into machine language (the machine language resulting from the translation process is stored on the RAD for use in phase two) and (2) secondly, the machine language is loaded and executed.

During phase 1 (the translation process) all of the Algol statements are read from punched cards and printed out on the High Speed Printer along with any diagnostic messages which the translator deems necessary and the machine language equivalent of the Algol program is written on the RAD. Note that the Algol program is not executed during this phase.

During phase 2 (the execution process) the machine language plus any necessary library routines (e.g. SIN or READ) are copied from the RAD into the computer memory and executed. It is during phase 2 that the data (if any) is read into the computer. The machine language which is executed during phase 2 is referred to as the object program.

The Algol translator is usually referred to as the compiler. Hence note that the word "compiler" refers to a program (i.e. a translating program) and not a collection of electronic devices.

1.3.1 DEVIATIONS FROM ALGOL 60

For those familiar with the Algol 60 publication language, the deviations of Sigma 5/9 Algol from Algol 60 can be summarized as follows:

1) Uniqueness of an identifier is determined by examining its first eight characters only.
2) Every formal parameter must be mentioned in the specification part of the procedure heading.
3) Numeric labels are excluded from the language.
4) A comma is the only acceptable parameter delimiter in procedure calls.
5) The result of integer exponentiation (I**J,I and J both integer) is always integer.

6) Forward referenced identifiers should be declared by a LOCAL declaration in those cases where the identifier may be confused with other global identifiers.
7) OWN arrays are dynamic and allocated only on the first use of the declaration.
8) Only one case of letters is provided.

Details concerning these and other restrictions are covered in more detail in other sections of this manual.

In addition extensions have been made to the Algol 60 language to facilitate the handling of large complex programs. These extensions can be summarized briefly as follows:

1) Input/Output routines have been defined to provide very flexible handling of the various data media.
2) Double precision and complex arithmetic has been implemented to extend the scope of scientific computations.
3) General string operations have been defined to provide very flexible data processing features.
4) Provisions have been made for allowing external procedures written in Algol or Metasymbol to be incorporated easily in the object program.
5) Options have been provided to facilitate debugging of the program.
6) An option has been provided to generate relocatable object code (ROM's) instead of using the special fast loader.
7) Bit operations on general expressions have been defined to provide flexible logical control.
8) A define macro has been added to provide compile time flexibility.
9) External variables and labels have been added to provide improved communication between external procedures and the main program.
10) An option has been provided to allow inline coded subscript computation.
11) An option has been provided to allow full bounds checking on subscript computation.

1.3.2 BASIC SYMBOLS

The following correspondences are made for the representation of basic symbols:

Basic Symbol for Reference Language	Basic Symbol for Translator
true	TRUE
false	FALSE
*	*
-	-
×	*
/	/
÷	//
↑	**
<	<
≤	LEQ or =
=	=
≥	GEQ or =
>	>
≠	NEQ or =
≡	EQIV
⊃	IMPL
∨	OR or !
∧	AND or &
¬	NOT or ¬
go to	GO TO
if	IF
then	THEN
else	ELSE
for	FOR
do	DO
,	,
.	.
10	@
:	:
;	;
:=	:=

Basic Symbol for Reference Language	Basic Symbol for Translator
step	STEP
until	UNTIL
while	WHILE
comment	COMMENT
(	(
)	)
[	(
]	)
‘	'
’	'
begin	BEGIN
end	END
own	OWN
Boolean	BOOLEAN
integer	INTEGER
real	REAL
array	ARRAY
switch	SWITCH
procedure	PROCEDURE
string	STRING
label	LABEL
value	VALUE

For the extensions to the language the following new basic symbols have been introduced:

define	DEFINE
external	EXTERNAL
format	FORMAT
go	GO
$\underline{\vee}$	XOR or ||
local	LOCAL

Basic Symbol for Reference Language	Basic Symbol for Translator
$	$ (substring operator)
¢	¢ or CAT (concatenation operator)
#	# (define terminator)
%	% (line termination character)

Typographical features such as "blank space" or "change to new line" have no significance in the translator's language except that blank space may not appear within basic symbols, identifier, and numbers. Blanks may, however, be used freely for facilitating reading.

CHARACTER SETS
IDENTIFIERS
QUANTITIES
VARIABLES
CONSTANTS
FUNCTIONS

II...

ELEMENTS OF THE
COMPILER LANGUAGE

2.1 CHARACTER SETS

The ALGEBRAIC COMPILER employs a character set which is commonly available as the extended 029 keypunch code. These characters are:

THE ROMAN ALPHABET

A, B,.....,Z

THE ARABIC NUMERALS

0,1,.....,9

SPECIAL CHARACTERS

+
-
=
(
)
.
,
@
/
*
<space>
'
:
&

SPECIAL CHARACTERS

<

>

;

¬

$

¢

#

%

In addition, some multiples of characters are given meaning as though they constituted a single character:

:=	replacement
**	exponentiation
¬ =	not equal
||	exclusive or
//	integer divide
> =	greater than or equal
< =	less than or equal
@@	double precision power of ten

2.2 IDENTIFIERS

The findamental construct of the compiler language is the identifier. Identifiers are used to name the various things which make up a program, for example, variables, functions, labels, procedures etc. An identifier is composed of a string of letters and digits. There is no limitation to the length of this string; however, the programmer should exercise care when choosing identifiers since the compiler considers two identifiers to be the same if they have the same first eight characters. The first character of an identifier must be a letter; no special characters (including spaces) may be embedded within an identifier.

In addition, a few identifiers are reserved for special use as operators, and punctuation marks.

These reserved identifiers may not be used by the programmer in any context other than that set down in this manual. A list of the reserved identifiers is given in APPENDIX C. Any other identifiers may be used at will; however, many other identifiers are predefined as library functions and should be avoided. In order to call your attention to these reserved identifiers they will be underlined whenever they appear in examples.

EXAMPLES:

Z
GAMMA
REAL
PARKAVENUESOUTH
A374
FUNGEKUTTAGILL
GEQ

2.3 QUANTITIES

The Compiler is concerned with the manipulation of six types of quantities_ real quantities, integer quantities, Boolean quantities, double precision quantities, string quantities, and complex quantities.

Real quantities represent the class of real numbers to an accuracy of seven significant decimal digits, the maximum permitted by the word length of the Sigma 5-9.

Integer quantities represent the class of integers that can be expressed in the word length of the Sigma 5-9, i.e., the integers whose magnitude is less than 2,147,483,647.

Double precision quantities represent the class of real numbers to an accuracy of sixteen decimal significant digits.

Boolean quantities represent a string of 32 logical balues where 1 represents TRUE and 0 FALSE. The quantity is considered TRUE if at least one of the 32 logical values is TRUE; otherwise FALSE.

String quantities represent a string of valid characters of arbitrary length less than 256.

Complex quantities represent the class of complex numbers. The real and complex parts are real numbers to an accuracy of seven significant decimal digits.

A program may contain quantities of any or all of these types. The programmer assigns the types of the variables, evaluated functions, and expressions which appear in his program. (See Chapter V).

2.4 VARIABLES

Variables treated by this compiler are of two kinds - simple variables and variables with subscript(s). A simple variable represents a single quantity and is denoted by the identifier which names the array, followed by a subscript list enclosed in parentheses. A subscript list consists of the arithmetic expressions separated by commas.

EXAMPLES:

Simple Variables

```
X
ALPHA
C13
```

Variables with Subscripts

```
A(I,J)
M(I+1,J+1)
V(F(P+1),12+Q)
Z(W(T),X(T),Y(T),Z(T))
C(13)
```

The expressions (See Chapter 111) which make up the subscripts of a variable with subscripts may be of any complexity. Real values are allowed, in which case the real number is rounded to the nearest integer. Each subscript expression must have a value which is not less than the minimum and not greater than the maximum specified for that array by the ARRAY declaration. The number of subscript expressions must equal the number of dimensions

of the array as declared in the ARRAY declaration for that array.

Whether a variable represents a real, integer, Boolean, real 2 (double precision), complex or string quantity is determined by the "declaration of type" described in Chapter V.

2.5 CONSTANTS

2.5.1 INTEGER CONSTANTS

Integer constants are represented by a string of digits. A maximum of nine significant digits is allowed. Leading zeros are ignored. Spaces may not be imbedded within an integer.

EXAMPLES:

```
0
17
16384
2111
```

2.5.1.1 HEXADECIMAL CONSTANTS

Hexadecimal constants are represented by a string of hexadecimal degits (0-9,A-F) enclosed by apostrophes immediately preceeded by the letter X. A maximum of eight hexadecimal digits are allowed including leading zeros. Hexadecimal constants are of type integer.

EXAMPLES:

```
X'41C00000'
X'40C8C5E7'
X'1FFFF'
```

2.5.1.2 BIT CONSTANTS

Bit constants are represented by a string of 0's and 1's enclosed by apostrophes immediately preceeded by the letter B. A maximum of 32 bits are allowed including leading zeros. Bit constants are type integer.

EXAMPLES:

B'1'

B'1111O'

2.5.2 REAL CONSTANTS

Real Constants are represented by a string of digits which contains "." - a decimal point. The decimal point may not appear at the end of the string. A real constant may contain a maximum of seven significant digits.

EXAMPLES:

3.141592

43.O

.6394

If desired, a scale factor amy be appended to a real constant to indicate that it is to be multiplied by the indicated power of 10. This scale factor is written as an @ followed perhaps by a + or - sign and then by an integer. The integer specified the power of 10 to be used, and is limited to a two-digit number.

EXAMPLES:

2.6@5 means 2.6 x 10^5 or 260,000.0

1.7@3 means 1.7 x 10^{-3} or 0.0017

A third option allows a real number to be written as an integer followed by a scale factor.

EXAMPLE:

3@4

This is precisely equivalent to writing 3.0@+4 or 30;000.0. Note that a scale factor alone may be used to specify a real number - for example 10^{-2} may be written as @-2, etc.

2.5.3 REAL 2 CONSTANTS (DOUBLE PRECISION)

Real 2 Constants are represented by a string of more than seven digits which contains "." - decimal point. A real 2 constant may contain a maximum of sixteen significant digits.

EXAMPLES:

0.00006174205

2.71828182845904

If desired, a scale factor may be appended to a real 2 or real constant to indicate both that it is to be multiplied by the indicated power of 10 and that it is double precision.

EXAMPLE:

1.0@@-1 or @@-1 or 1@@-1

This represents 0.1 correct to sixteen significant figures. One should note that 0.1 is not represented exactly in binary, and that double precision representation will be more accurate than the normal single precision.

2.5.4 BOOLEAN CONSTANTS

Only two Boolean Constants are allowed - TRUE and FALSE

2.5.5 STRING CONSTANTS

String constants are represented by any string of acceptable characters (excluding an apostrophe) enclosed by apostrophies. (Note: An apostrophe may be entered into the string by using double apostrophies ('').)

EXAMPLES:

'128F6.2'

'HOW␣NOW␣BROWN␣COW'

'JOHN␣BROWN''S␣HOUSE'

where ␣ denotes a blank (space). The maximum length allowed is 255 characters. Also, zero-length strings are not allowed.

2.5.6 COMPLEX CONSTANTS

Complex constants are represented by a real or integer constant immediately followed by the letter I. The real part is set to zero.

The real and imaginary part of the complex constant are of type real.

EXAMPLES:

51

3.6I

2.7@2I

2.6 FUNCTIONS OR EVALUATED PROCEDURES

The compiler allows the use of a wide variety of functions. In this section we will consider only the simplest form of functional notation in order to provide a basis for the next chapter. (Chapter X contains a complete description of the use of procedures and the manner in which they are defined.) For the moment we will assume that a procedure acts on one or more quantities called arguments and produces a single number as a result. This resulting quantity is called a functional value.

GENERAL FORM:

$$\xi(\epsilon_1,\ldots,\epsilon_q)$$

where ξ is an identifier which names the procedure and ϵ_1 throough ϵ_q are expressions which are the arguments of the procedure.

EXAMPLES:

```
SIN(X)
SQRT(B**2-4*A*C)
HYPERGEOM(A,B,C,Z)
LN(SIN(THETA-ALPHA/2))
```

The type of a procedure depends on the manner in which the procedure was defined. The type required for each of the arguments is also determined by the definition of the procedure. In general, it is the programmer's responsibility to ensure that each of the arguments is of the proper type. However, parameters will have the arithmetic converted if possible, and all other violations will cause an error message. The parameters to library procedures will be converted to the proper type whenever possible.

ARITHMETIC EXPRESSIONS
BOOLEAN EXPRESSIONS
STRING EXPRESSIONS
BIT EXPRESSIONS
DESIGNATIONAL EXPRESSIONS

III...

EXPRESSIONS

Algol statements deal with five kind of expressions: Arithmetic expressions (those having numerical values), Boolean expressions (those having truth values), Designational expressions (those having statement labels as values), String expressions (those having strings as values), and Bit expressions (those referencing a sequence of bits). This chapter describes the manner in which these expressions may be combined to produce new expressions. Expressions must be well formed in accordance with mathematical convention and with the rules set forth below.

3.1 ARITHMETIC EXPRESSIONS

Arithmetic quantities are combined by means of the operations +,-,*,/,//, and **. The symbol ** is used to denote exponentiation, (↑), i.e., B**2 has the meaning B^2 or (B↑2), and // denotes integer divide* (÷). In addition to these six symbols, parentheses may be employed to indicate that a specific order of evaluation is to be followed rather than the assumed order in ALGOL. To be explicit, it is assumed -- in the absence of parentheses to indicate otherwise -- that exponentiation is performed before multiplication and division, multiplication and division before addition and subtraction. Operations on the same level (e.g., addition and subtraction) are done from left to right. Parentheses should be used to express the exact meaning desired.

In mixed mode expressions, the operands are converted to the most general arithmetic type. Starting with the least general, the types are ordered as: STRING, INTEGER, REAL, REAL 2, COMPLEX.

*Note: The // operator requires that both operands be integer expressions and gives an integer result. The/ operator, however, uses operands of any arithmetic type and gives a result of the corresponding type except in the integer case which yields a real result.

3.1.1 THE CONDITIONAL ARITHMETIC EXPRESSION

Another class of arithmetic expressions is comprised of those which result from a test on a Boolean expression.

GENERAL FORM:

$$\underline{\text{IF}}\ \beta\ \underline{\text{THEN}}\ \varepsilon_1\ \ \underline{\text{ELSE}}\ \varepsilon_2$$

where ε_1 and ε_2 are arithmetic expressions and β is any Boolean expression. If β has the value TRUE, then the value of this arithmetic expression is the value of ε_1; otherwise the arithmetic expression has the value of ε_2. ε_1 may not be a conditional arithmetic expression unless it is enclosed in parentheses. Moreover, it is necessary to enclose the whole conditional arithmetic expression in parentheses whenever it appears within a larger expression.

EXAMPLES:

```
IF I<O THEN N ELSE N-1
IF A>O THEN U+V ELSE IF B=O THEN B/A ELSE Z
IF Q<O THEN (IF A* 17 THEN U/V ELSE V/U) ELSE O
A* (IF B>O THEN C ELSE D)/E
```

3.1.2 STRINGS IN ARITHMETIC EXPRESSIONS

Whenever a string quantity (variable, expression, constant, or procedure) is used within the context of an arithmetic expression, then the string is assumed to be a string of digits optionally preceded by an arbitrary number of blanks followed by an optional '+' or '-' sign, and is converted automatically to an integer quantity denoting the value of the string. If the assumption of a string of digits is false, an error at run time will be detected and an error message will be indicated.

As an example, let:

A be the string '4'
B be the string '+9Δ '

then A*B has the value 36
and A**'2' has the value 16.

Note: '2' + '36' is equivalent to 2+36 as an arithmetic expression.

3.1.3 CONSTRUCTION OF ARITHMETIC EXPRESSIONS

A variable, a constant, or an evaluated procedure of real, integer, complex or real 2 type will in itself constitute an arithmetic expression. Furthermore, if we let ε_1 and ε_2 represent any arithmetic expression and let β represent any Boolean expression, then each of the following combinations is also an arithmetic expression:

$\varepsilon_1 * \varepsilon_2$	$\varepsilon_1 + \varepsilon_2$
$\varepsilon_1 / \varepsilon_2$	$\varepsilon_1 - \varepsilon_2$
$\varepsilon_1 ** \varepsilon_2$	$+\varepsilon_1$
(ε_1)	$-\varepsilon_1$
$\varepsilon_1 // \varepsilon_2$	IF β THEN ε_1 ELSE ε_2

EXAMPLES:

Expression	Compiler Interpretation
A+B*C	A+(B*C)
A*B+C	(A*B)+C
A/B*C	(A/B)*C
A*B/C	(A*B)/C
-X*Y	(-X)*Y
-X-Y	(-X)-Y

3.2 BOOLEAN EXPRESSIONS

Boolean quantities may be combined by means of operations to form Boolean expressions in a manner entirely analogous to the combination of arithmetic quantities by arithmetic operations. Boolean expressios are again true or false, depending entirely on the truth values of the quantities entering into the expression and the definitions of the Boolean operations combining them.

The Boolean operators which are accepted by the compiler are ¬, &, |, ||, IMPL, and EQIV. These operations are called negation, conjunction, disjunction, exclusive disjunction, implication, and equivalence, and are defined as follows (P and Q are Boolean quantities):

P	Q	¬P	P & Q	P | Q	P || Q	P IMPL Q	P EQIV Q
false	false	true	false	false	false	true	true
true	true	false	true	true	false	true	true
true	false	false	false	true	true	false	false
false	true	true	false	true	true	true	false

Conventions for the order of precedence of Boolean operations are not as well established as are those for arithmetic operations. However, we shall assume the following order, which is apparently the most common:

Unless indicated other wise by the use of parentheses, ¬ will be executed before &, & will be executed before | and || ; | and || will be executed before IMPL; and IMPL will be executed before EQIV. In the case of equal priorities, operations are executed from left to right. For example, P IMPL Q IMPL R means ((P IMPL Q) IMPL R).

EXAMPLES:

```
¬(P ξ Q) | R IMPL P | Q
¬ (¬P) EQIV P
 P IMPL P | U ξ V
 (P | Q) ξ ¬(P ξ Q)
 (A <= X) ξ X <= B
 (ERROR< TOLERANCE) || (N > 40)
R ξ S | (F(X)  =  4)
(M*N(R - 2) + 4 < SIN(BETA - ALPHA)) | FLAG
```

3.2.1 RELATIONS

Another class of Boolean expressions is comprised of those which result from a test on arithmetic expressions. These are termed arithmetic relations, and consist of two arithmetic expressions and a relational operator. The latter is an operator in the sense that it performs a transformation on the comparison to produce a truth value. This value is either true or false depending upon the results of the comparison.

GENERAL FORM:

$$\varepsilon_1 \rho \varepsilon_2$$

where ε_1 and ε_2 are arithmetic expressions, and ρ is a relational operator.

This relation has the value true if ε_1 is in the relation ρ to ε_2; it is otherwise false.

The relational operators employed in the compiler are >, >=, =, <=, <, and ¬=. Their significance is indicated in the following table.

EXPRESSION	CONVENTIONAL MATHEMATICAL NOTATION	MEANING
$\varepsilon_1 > \varepsilon_2$	$\varepsilon_1 > \varepsilon_2$	greater than
$\varepsilon_1 >= \varepsilon_2$	$\varepsilon_1 \geq \varepsilon_2$	greater than or equal to
$\varepsilon_1 = \varepsilon_2$	$\varepsilon_1 = \varepsilon_2$	equal to

EXPRESSION	CONVENTIONAL MATHEMATICAL NOTATION	MEANING
$\varepsilon_1 <= \varepsilon_2$	$\varepsilon_1 \leq \varepsilon_2$	less than or equal to
$\varepsilon_1 < \varepsilon_2$	$\varepsilon_1 < \varepsilon_2$	less than
$\varepsilon_1 \neg = \varepsilon_2$	$\varepsilon_1 \neq \varepsilon_2$	not equal to

EXAMPLES:

```
X¬ = 0
(ABS(L - LPRIME) < EPSILON)
T > TMAX
```

3.2.1.1 STRINGS IN RELATIONS

Strings may be compared by the above relational operators. The comparisons are made using the natural collating sequence of characters on the SIGMA 5/9 (i.e., EBCDIC code).

A string comparison is made only if both sides of the relation are strings. If either operand is non-string then the string operand is converted automatically to its associated arithmetic expression before performing the comparison as in the arithmetic expression case.

If the strings are not of the same length, then the shorter string is extended to the length of the longer string by appending blanks.

3.2.1.2 COMPLEX QUANTITIES IN RELATIONS

Only = and ≠ are valid relational operators when the operands are complex quantities.

3.2.2. THE CONDITIONAL BOOLEAN EXPRESSION

A Boolean expression can also be made subject to a condition.

GENERAL FORM:

$$\underline{\text{IF}}\ \beta\ \underline{\text{THEN}}\ \beta_1\ \underline{\text{ELSE}}\ \beta_2$$

where β, β_1, and β_2 stand for Boolean expressions. The evaluation of the conditional Boolean expression is the same as that of the conditional arithmetic expression (Sec. 3.1.1).

EXAMPLES:

```
IF K<1 THEN X=-Z ELSE P&Q|W Y
IF IF A THEN B ELSE C THEN D ELSE F
P&(IF K<1 THEN S>W ELSE H<C)|Q
```

3.2.3 CONSTRUCTION OF BOOLEAN EXPRESSIONS

Any variable, constant, evaluattd function, or procedure will itself constitute a Boolean expression, if it is of Boolean type.

In addition, if $\varepsilon_1\ \rho\ \varepsilon_2$ is an arithmetic relation, and β_1, β_2 and β_3 are any Boolean expression, then each of the following is also a Boolean expression:

$(\varepsilon_1\ \rho\ \varepsilon_2)$	$\beta_1 \mid \beta_2$	(β_1)
β_1	β_1 EQIV β_2	$\beta_1 \mid\mid \beta_2$
$\beta_1 \xi \beta_2$	β_1 IMPL β_2	β_3 THEN β_1 ELSE β_2
		IF $\varepsilon_1\ \rho\ \varepsilon_2$ THEN β_1 ELSE β_2

3.3 STRING EXPRESSION

The partial string ($) operator is used to denote a partial string variable or expression. As an example, consider S to be a string variable then

```
S$(I)
```

denotes the Ith character in the string S, where the characters are numbered from the left starting with one. Thus, S$(6) is the sixth character in the string S. If we write

```
S$(I,J)
```

then this expression denotes a string variable of J characters taken in ascending order from string S starting with the character in the Ith position. Also, the ($) operator can

be used with any expression of stype STRING to denote a partial string expression in a similar fashion.

The concatenation operator (c or CAT) is used to concatenate two string expressions to form an expression of type STRING. Assuming S and T variables of type STRING, then

```
S&T
```

is a string expression consisting of the characters in S followed by those in T.

Note: String values resulting from concatenation operations or string procedure calls should not be passed as arguments nor returned as procedure values.

3.4 BIT EXPRESSIONS

The bit (.) operator may be used with an expression of any type to extract partial bit fields. The resulting expression is of type INTEGER. As an example, consider

```
B.(I,J)
```

Then this expression selects J bits taken in ascending order from variable B starting with the bit in the Ith position, where the bits are numbered from the left starting with one. (Note: Since the result is of type integer, the maximum value that J can have is 32).

EXPRESSION TYPE	BIT NUMBERING (Left to Right)
BOOLEAN	1 to 32
INTEGER	1 to 32
REAL	1 to 32
REAL 2	1 to 64
STRING	1 to (8 * length of string)
COMPLEX	1 to 64

3.5 DESIGNATIONAL EXPRESSIONS

Designational expressions are those expressions whose values are statement labels. The form of a designational expression is either a label, a switch variable or a

conditional expression between designational expressions.

For example, if d_1 and d_2 are designational expressions, β is a Boolean expression, L a statement label, S a switch variable and ε an arithmetic expression, then each of the following is a designational expression:

L

S(ε)

IF β THEN d_1 ELSE d_2

3.6 PRECEDENCE OF OPERATORS

The sequence of operations within one expression is generally from left to right, with the following rules of precedence:

First:	unary minus, unary plus	
Second:	$\not\in$	
Third:	arithmetic expressions	First: ** Second:*,/,// Third: +,-
Fourth:	$<, \leq, =, \geq, >, \neg=$	
Fifth:	$\neg$	
Sixth:	ξ	
Seventh:	$\mid, \parallel$	
Eighth:	IMPL	
Ninth:	EQIV	

An expression contained in parenthesis is evaluated by itself and this value is used in any subsequent evaluation. Therefore, the desired order of execution of operations within an expression can always be effected by appropriate positioning of parentheses.

ASSIGNMENT STATEMENTS
GRAMMAR OF STATEMENTS
COMPOUND STATEMENTS
STATEMENT LABELS
DUMP STATEMENTS
COMMENT

IV...

STATEMENTS

The statement, , is the fundamental unit of expression in the description of an algorithm. Most of what follows in this manual deals with the formation of statements and their interrelation to form larger constructs. Statements may be divided into two classes -- the operational statement and the declarative statement. Operational statements specify something that the object program is to do. Declarative statements give information to the compiler about the program being compiled. After this chapter, the word statement will usually be employed to mean an operational statement; a declarative statement will then be called a declaration. However, for the present, "statement" will stand for either sort.

The first part of this chapter discusses one particular kind of operational statement -- the assignment statement. The last part of the chapter deals with the grammer of statements in general, using assignment statements for examples.

4.1 THE ASSIGNMENT STATEMENT

The assignment statement specifies an expression which is to be evaluated and a variable which is to have the resulting value assigned to it.

GENERAL FORM:

$$\tau := \epsilon$$

where τ is a variable and ϵ is an expression. Note that the symbol := is used in a special sense in this compiler to signify the process of substitution. Thss X:= X + 1 means "using the current value of the variable X, evaluate the expression X + 1, and assign the result as the new value of X."

4.1.1 ARITHMETIC ASSIGNMENT STATEMENTS

If the variable Γ:= aϵ is INTEGER, REAL, REAL 2, or COMPLEX type, then we have an arithmetic assignment statement and a denotes an arithmetic expression. If Γ and aϵ are of different types then aϵ will be converted to the type of Γ if possible before the assignment is made. If the conversion is from real to integer then the result is rounded to the nearest integer.

EXAMPLES:

```
R:= (-B + SQRT(B**2 - 4*A*C2))/(2*A)
U:= X*COS(X*COS(THETA) + Y*SIN(THETA))
OMEGA:= 1/SQRT(L*C)
E:= M*CC**2
C(I,J):= C(I,J)+ A(I,K)*B(K,J)
```

4.1.2 STRING ASSIGNMENT STATEMENTS

If the variable Γ in Γ:= Sϵ is a string variable, we have a strin g assignment statement. In this case, the expression Sϵ must be either arithmetic or of type string. If Sϵ is an arithmetic expression then it is first converted into a signed integer and then into its associated string with a leading '-' sign if negative.

In all cases the replacement is made such that the left most character of the right hand side replaces the left most character in the left hand string variable. Extra spaces are supplied to the right as necessary to fill out the left hand string and any excess of characters from the right hand side will be dropped. (ie., The replacement is left justified in the left hand string variable).

As an illustration, consider the following uses in which A is a string variable:

A BEFORE	STATEMENT	A AFTER
ABCDEF	A:='XYZUVW'	XYZUVW
ABCDEF	A:='LOOP-DE-LOOP'	LOOP-D
ABCDEF	A:='HOW'	HOW
ABCDEF	A$(2):='Q'	AQCDEF
ABCDEF	A$(2,3):='XYZ'	AXYZEF
ABCDEF	A$(2,3):=69	A69 EF
ABCDEF	A:=-3.1415927	-3▲▲▲▲

EXAMPLES:

```
S:=1000*SIN(X)
S:=IF X > 1 THEN 'P1' ELSE 'P2'
S$(1,N):=S$(2,N-1)
```

One word of caution, the string replacements are performed a character at a time starting with the left most character- hence, a replacement of the form;

```
S$(2,N-1) := S$ (1,N-1)
```

will result in the character in the 1,1 position, S$(1,1), being propagated down the string (i.e., the first N characters of the string S will all be the same as the character in S$(1,1)). In order to shift the characters in a string right, it is necessary to first move the string into another string of the same length. For example, let S and T be strings of the same length then a right shift of one place can be performed on S by:

```
T:= S;
S$(2,N-1):= T$(1,N-1)
```

4.1.3 BIT ASSIGNMENT STATEMENTS

If the variable Γ in Γ:= aϵ is a bit expression, we have a bit assignment statement. In this case, the expression aϵ will be converted to integer if necessary.

In all cases, the replacement is made such that the right most bit of the right hand side replaces the right most bit in the left hand bit expression. Extra zeros are supplied to the left as necessary to fill out the field and any excess bits from the right hand side will be dropped.

(i.e., The replacement is right justified in the left hand bit expression.)

If in Γ, the operand of the bit operator is a variable of type BOOLEAN, then the value of Γ considered as a logical value is TRUE if $\Gamma \neq 0$ and FALSE if $\Gamma = 0$.

As an example, consider that if I, an integer variable, has initially the value 14(i.e., 01110 in binary) then

1.(28,2):=2

causes the value of I to be changed to 22 (i.e., 01011O in binary).

4.1.4 BOOLEAN ASSIGNMENT STATEMENTS

If the variable Γ in $\Gamma := \beta\varepsilon$ is a Boolean variable, we then have a Boolean assignment statement. In this case, the expression $\beta\varepsilon$ must be Boolean.

EXAMPLES:

FLAG:=(SWITCH4|SWITCH5) ξ FLAGPRIME
TEST:= (X¬=0) ξ (Y¬ = 0)
TOGGLE3:= TOGGLE4 ξ TAG | (U<V)

4.1.5 GENERALIZED ASSIGNMENT STATEMENT

GENERAL FORM:

$$\Gamma_1 := \Gamma_2 := \ldots \Gamma_N := \varepsilon$$

If it is desired to assign the same value to a number of variables, it can be accomplished in a single statement by employing this generalized form.

Note that if the list of variables to which a value is being assigned is mixed type, then conversion of type will be performed- e.g., assume X,Y, and ε are real, and I is integer. Then the statement

X:= I:= Y:= ε

will cause ε to be stored into Y, rounded to an integer before storing into I, and then this rounded result to be stored into X. Thus, for the above conditions, X:= I:= Y:= ε,

X:= Y:= I:= ε, and I:= X:= Y:= ε may all give different results when ε is real.

EXAMPLES:

```
V:= X:= Y:= 15.302
A(I):= B(I):= Z:=0
```

4.2 THE GRAMMAR OF STATEMENTS

This section discusses certain definitions and rules of the compiler language which have to do with the writing of statements. The basic rule of the grammar of statements is that a statement must be separated from a following statement by a semicolon.

Even though a statement ends on a given line and the next statement begins on the next line, the separating semicolon must be indicated. The end of a line has no meaning as punctuation.

GENERAL FORM:

$$\Sigma;\Sigma;\cdots\Sigma$$

where the symbol Σ represents any statement. Unless otherwise indicated, statements are pefformed one after the other in the sequence in which they are written. As many statements as desired may be written on a line (subject of course to the physical limitations of the input medium), or a statement may use as many lines as are required for its expression.

EXAMPLE:

```
W:=A+B; X:= A-B; Y:= A*B; Z:=A/B
```

4.3 COMPOUND STATEMENTS

It is frequently desirable to group several statements together to form a larger construct which is to be considered as a single statement. Such a construct is called a compound statement.

GENERAL FORM:

BEGIN $\Sigma_1;\Sigma_2;\cdots\Sigma_N$ END

Where Σ_1 through Σ_N are statements. The words BEGIN and END serve as opening and closing statement parentheses.

Throughout this description of the compiler, unless the contrary is specifically stated, the word "statement" and the symbol Σ should be construed to mean either a simple or a compound statement.

Certain other constructs involving the grouping of several statements automatically constitute compound statements. These will be discussed further in their proper context in Chapter VI.

EXAMPLES:

```
BEGIN U:=-B/(2*A):V:=SQRT(U**2-C/A);
R1:= U+V; R2:= U-V END

BEGIN S:= SIN(THETA); C:= COS(THETA);
X1:= C*X + S*Y; ETA:= -S*X + C*Y END
```

4.4 STATEMENT LABELS

It is often necessary to attach a name to a statement (e.g., if one wishes to get from the end of a program back up to the beginning). This name is called a statement label. A statement label must be an identifier.

GENERAL FORM:

$$a : \Sigma$$

where a is an identifier, and Σ is any statement.

EXAMPLES:

```
START:    SUM:=0
LEGENDRE:P(N):= ((2*N - 1) *P(N)-1) - (N-1)* (N-2))/N
ROTATE:   BEGIN S:= SIN(THETA); C:= COS(THETA);
          X1:= C*X + S*Y; ETA:= -S*Y + C*Y END
```

When using a compound statement, the programmer may insert a comment after the word END. The comment is terminated by the next ";", END, or ELSE. This may be done for readability of the print-out produced during compilation; the compiler itself makes no use of the information. However, if it is dubious whether or not a comment is intended, a warning message is given.

GENERAL FORM:

a: BEGIN $\Sigma_1; \Sigma_2; \cdots \Sigma_N$ END sequence of symbols not containing a ";", END, or ELSE

EXAMPLE:

```
ROOTS: BEGIN U:=-B/(2*A);V:=SQRT(B**2-4*A*C)/(2*A);
R1:= U+V;R2:= U-V END ROOT PROCESSING SECTION
```

4.5 DUMP STATEMENT

The DUMP Statement has been added to the language to provide the programmer with a selective post-mortum dump when an error is detected during the running of the object program.

GENERAL FORM:

DUMP <dump list>

where <dump list> is a sequence of <dump list elements> separated by commas. The <dump list elements> may be any of the following quantities:

Constants	The constant may be of type BOOLEAN, STRING, INTEGER, REAL, REAL2, or COMPLEX.
Identifiers	The identifiers must be local to the block and not a call-by-name parameter. They may be either a simple variable or an array identifier. Any of the allowable arithmetic types (i.e., BOOLEAN, STRING, INTEGER, REAL, REAL2, or COMPLEX) are permitted.

The effect of the DUMP statement on a program is that when an error is detected in the running program, the < dump list > associated with the last executed DUMP statement in each block is printed before terminating the program.

4.6 THE COMMENT

The COMMENT allows the programmer to include any clarifying remarks, identifying symbols, etc., in the printed compilation. The COMMENT does not appear as part of the compiled program, and has no effect on the program; it merely sets apart any string of characters for printing as part of the compilation. Since the comment extends to the next semicolon, a semicolon obviously cannot be used within the string of characters.

```
COMMENT GOOD GRIEF!!!
```

The "%" Line terminator causes the scanning operation to terminate at this point in the record and to proceed to the next record. All information after the "%" line terminator in the record will be treated as a comment and ignored. This feature permits comments to be inserted at the end of a record.

```
X:= 0; % INITIALIZE THE VARIABLES
```

TYPE	SWITCH		
STRING	LOCAL	V...	
ARRAY	DEFINE		
OWN	EXTERNAL		BASIC DECLARATIONS

The Declarations of Type -- REAL, INTEGER, BOOLEAN, COMPLEX, and STRING are defined in this chapter, together with the ARRAY, OWN, SWITCH, LOCAL, EXTERNAL, and DEFINE declarations These do not exhaust the entire set of declarations available to the programmer; however, the others constitute separate subjects in themselves and are therefore reserved for later chapters.

Declarations determine how the compiled program will treat certain of its elements. It is thus necessary to precede the use of an identifier with a declaration of its type.

5.1 DECLARATIONS OF TYPE

Declarations of type are used to indicate that a specified set of identifiers represent quantities of a given type (REAL, REAL2, INTEGER, BOOLEAN, COMPLEX, or STRING).

CONSTRUCTION OF DECLARATIONS OF TYPE

GENERAL FORM:

REAL	⟨type list⟩
REAL 2	⟨type list⟩
INTEGER	⟨type list⟩
BOOLEAN	⟨type list⟩
COMPLEX	⟨type list⟩
STRING	⟨type list⟩

These statements declare the identifiers given in ⟨type list⟩ to be single precision REAL, double precision REAL, INTEGER, BOOLEAN, COMPLEX and STRING type respectively. The integer "2" following the declaration REAL is used to denote double precision type for the identifiers in ⟨type list⟩ .

In general, a < type list > consists of a sequence of identifiers separated by commas.

EXAMPLE:

INTEGER I, J, K, L, Z

5.2 THE STRING DECLARATION

The STRING declaration provides a means of referring to a collection of alphanumeric characters in EBCDOC code by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection. It is also possible to subdivide this collection into pieces, each of which may be assigned a different identifier. In effect, this means that string variables may be partitioned into a set of substrings, each with a different identifier. This partitioning may be nested to any depth.

5.2.1 CONSTRUCTION OF STRING DECLARATIONS

A string variable must be described by a STRING declaration before the string variable is used.

GENERAL FORM:

STRING <string element> .----, <string element>

The < string element > 's take one of the following two forms:

FIRST FORM:

$\alpha(\sigma)$ OR α

where α is the name of the string and σ is an arithmetic expression which defines the numbers of characters in the string. If the specification of size is omitted after the name, then the specification will be assumed to be the same as that of the following string in the same declaration.

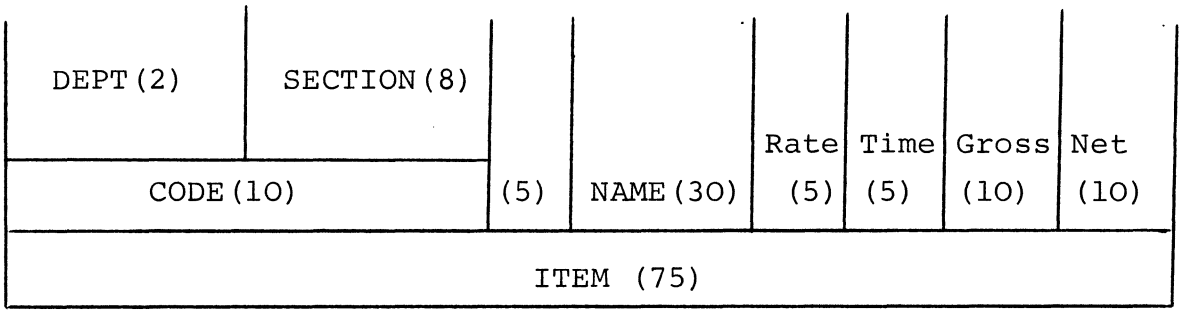
SECOND FORM:

$$\alpha := S\varepsilon$$

where α is the name of the string and $S\varepsilon$ is any string name, string constant or a substring operation on a string name or string constant. In this case the name α is defined to represent the string expression. If the expression is a string constant then α will be defined to be the length of the constant and the value of the constant will be the initial value of the string.

```
STRING  CARD(80),LINE(132),ITEM(75),CODE:=ITEM$(1,10),
        DEPT:=CODE$(1,2),SECTION:=CODE$(3,8),
        NAME:=ITEM$(16,30),RATE:=ITEM$(46,5),
        TIME:=ITEM$(51,5),GROSS:=ITEM$(56,10),
        NET:=ITEM$(66,10);
```

The string CARD holds 80 characters corresponding to a card image. Correspondingly, the string LINE holds one printer line image. The string ITEM, on the other hand, has the somewhat complicated structure shown below:



5.3 THE ARRAY DECLARATION

The ARRAY declaration provides a means of referring to a collection of elements (numbers or strings) by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection.

Arrays in this compiler are restricted to those of rectangular construction in n-dimensional space.

If the identifier of an array requires a declaration of type, then that declaration of type must precede the word ARRAY in the array declaration (cf. GENERAL FORM).

5.3.1 CONSTRUCTION OF ARRAY DECLARATIONS

An array must be described by an ARRAY declaration prior to the use of any variable with subscripts which represent an element of that array.

GENERAL FORM:

⟨type⟩ARRAY⟨array element⟩ ...., ⟨array element⟩

where

⟨type⟩ is any of the possible type declarations (REAL, REAL 2, INTEGER, COMPLEX, STRING or BOOLEAN). If the type declaration is omitted then REAL will be assumed. The ⟨array element⟩ 's are list items of the array declarator list. These list items take on the form:

$$\alpha(\lambda_1:\mu_1,\lambda_2:\mu_2,\cdots\lambda_N:\mu_N) \text{ or } \alpha$$

Where α is the name of the array, and each $\lambda_i : \mu_i$ pair are arithmetic expressions which represent the lower and upper bound on the i^{th} dimension, respectively. If the specifications of size are omitted after the name, then the specifications will be assumed the same as those of the following arrays. The length of a particular dimension can be calculated from $\mu_i - \lambda_i + 1$

EXAMPLE:

```
INTEGER ARRAY M,N(1:3,1:4),CHAR(-1:4,0:5,1:6),
                            VECTOR(I:J)
```

This declaration reserves twelve cells in storage for each of the two-dimensional arrays M and N, 216 cells for the three-dimensional array CHAR, and J - I + 1 cells for the one-dimensional array VECTOR.

5.3.2 THE STRING ARRAY DECLARATION

When dfining a string array, the general form of the array declaration must be modified to include the length of the string portion along with the partitioning of the string array into substring arrays.

GENERAL FORM:

STRING ARRAY ⟨string array element⟩,...,⟨string array element⟩

The ⟨string array element⟩'s take one of the following two forms:

FIRST FORM:

$$\alpha(\lambda_1 : \mu_1, \lambda_2 : \mu_2, \ldots, \lambda_N : \mu_N)(\sigma)$$

where α is the name of the string array, σ is an arithmetic expression which defines the number of characters for each element in the string array, and each $\lambda_i : \mu_i$ pair has the same interpretation as in the previous case.

Again as in the case of strings, it is possible to partition the string array into many substring arrays using the second form.

SECOND FORM:

$$\alpha := \alpha_1(s, \ell)$$

where α is the name of the string array having the same dimensions as the previously defined string array α_1 . The elements of α are defined as substrings of the elements of α_1 starting at position S and of length ℓ.

EXAMPLE:

```
STRING ARRAY A(1:6,-3:2)(20),B:=A$(11,10);
```

This declaration defines a string array A, each of whose elements consists of 20 characters, as a two-dimensional array. In addition, A is partitioned into three pieces, two of which are substring arrays B of 7 characters and C of 10 characters respectively. It should be noted that B and C will also be two-dimensional arrays.

5.4 THE OWN DECLARATION

All of the above declarations will assign space dynamically from local variable storage as needed. When the declarations are no longer in force, this space is returned to the program for other uses. If it is desired to permanently assign the space for a variable, each of the above declarati-ns must be prefixed by the word OWN. This form of declaration then becomes an OWN declaration. (See Chapter IX for further details)

EXAMPLES:

```
OWN INTEGER, I, J, K
OWN STRING  S(100)
OWN ARRAY FACTORIAL (0:30)
```

5.5 THE SWITCH DECLARATION

The SWITCH declaration provides a means of selecting a label, switch variable or designational expression from a list by means of a subscript expression on a switch identifier. In effect, the switch declaration defines a switch variable which is similar to a one-dimensional array except that the elements are labels, switch variables or designational expressions.

A switch must have been described by a SWITCH declaration prior to the use of any switch variable with subscripts which represents an element of the switch. The range of subscripts is from 1 to N, where N represents the number of elements in the switch. If a subscript expression on a switch variable falls outside the defined range of the switch then the switch operation is ignored.

The form of a SWITCH declaration is as follows:

GENERAL FORM:

$$\underline{\text{SWITCH}}\ \alpha := \varepsilon_1, \varepsilon_2, \cdots \varepsilon_N$$

where α is the name of the switch and each ε_i is a label, switch variable or designational expression representing a statement label.

EXAMPLE:

```
SWITCH S:=L1,IFX > Y THEN L2 ELSE L3,L4,T(1+6),L5
```

If the switch variable S is referenced from a GO TO statement (see chapter VI) as S(J), then the following transfer of control is made depending upon the value of J.

1) if J = 1 then control transfers to L1
2) if J = 2 then control transfers to either L2 or L3 depending upon X and Y.
3) if J = 3 then control transfers to L4
4) if J = 4 then switch T takes charge.
5) if J = 5 then control transfers to L5
6) if J < 1 or J > 5 then no transfer is executed.

5.6 THE LOCAL DECLARATION

Under certain conditions in a program, the necessity arises for using an identifier before it has been properly declared (e.g., using a label in a GO TO statement before the occurrence of the label definition). Such a reference to an identifier is said to be a forward reference.

In order for the compiler to properly treat all such identifiers, each identifier must be defined by a LOCAL declaration before it is used in the forward sense. The LOCAL declaration may be omitted for simple labels if the identifier has not occurred in any previous declaration. This declaration, in effect, localizes the identifier to the proper block (e.g. Chapter IX), as well as supplying information with regards to the type of identifier.

GENERAL FORM:

LOCAL < identifier type > $\alpha_1, \alpha_2, \ldots, \alpha_n$

where α_1 are identifier names and < identifier type > is any one of the following:

LABEL

PROCEDURE or < type > PROCEDURE

SWITCH

EXAMPLES:

```
LOCAL LABEL L1, L2, A6, BOX
LOCAL INTEGER PROCEDURE P1, P2
LOCAL SWITCH S1, S2, S3
```

5.7 DEFINE DECLARATION

GENERAL FORM:

DEFINE ⟨definition list⟩

where ⟨definition list⟩ is a sequence of definition parts separated by commas. Each ⟨definition part⟩ has the following form:

⟨defined identifier⟩ := ⟨well-formed construct⟩#

where ⟨defined identifier⟩ is the identifier associated with the ⟨well-formed construct⟩ which consists of any meaningful sequence of symbols from the language.

EXAMPLES:

```
DEFINE RK:= RUNGEKUTTA#, ROOT:= (-B+SQRT(B**2-4*A*C))/(2*A)#
DEFINE INT:= INTEGRATE (X,Y,Z)#
DEFINE LP:= (#,RP:=) , RTDIG:=(42:6)#
DEFINE FORI := FOR I := 1STEP 1 UNTIL #
```

SEMANTICS

The DEFINE declaration provides a method whereby an identifier can be defined to represent a well-formed ALGOL construct.

The appearance of a defined identifier in a program is equivalent to the appearance of its definition, and must result in a syntactically correct ALGOL construct.

At declaration time, a definition is of consequence; it has meaning only in relation to the context in which its related defined identifier appears. For this reason, undeclared identifiers may appear in definitions; all identifiers must have been declared, however, when the defined identifier is used.

During compilation, syntax errors (if any) in a definition are noted following the use of the defined identifier.

5.7.1 NESTING OF DEFINITIONS

Definitions can be nested; that is, defined identifiers may be used in definitions. For instance, in the example below, the definition for D3 is equivalent to the definition for DD. In the example, the definition +A+A is considered nested one level in the first declaration. In the second declaration the definition +A+A is considered nested two levels, etc.

EXAMPLE:

```
DEFINE D1  := +A+A#
DEFINE D2  := D1 D1#
DEFINE D3  := D2 D2#
DEFINE DD  := +A+A  +A+A  +A+A  +A+A#
```

RESTRICTIONS

A definition cannot be nested more than eight levels. Defined identifiers may not be used in a FORMAT declaration or STRING constant.

5.8 THE EXTERNAL DECLARATION

GENERAL FORM:

EXTERNAL ⟨other declarations⟩

where other declarations is a REAL, INTEGER, BOOLEAN, COMPLEX LABEL, PROCEDURE, STRING, or ARRAY declaration.

EXAMPLES:

```
EXTERNAL REAL X, Y, Z
EXTERNAL LABEL L1, L2
EXTERNAL INTEGER ARRAY A, B(1:10)
```

SEMANTICS:

This declaration, which cannot be used with the FL option, defines variables which can be referenced in any other separately compiled procedure that declares the same variables in EXTERNAL declarations. EXTERNAL variables are treated like OWN variables. An EXTERNAL declaration in the main

program causes the space to be allocated for the variables in OWN storage. In separately compiled procedures, the declaration causes an external reference to be generated for the variables. It is the user's responsibility to insure that the attributes of EXTERNAL identifiers match.

INTERNAL PROCEDURES
EXTERNAL PROCEDURES

X... PROCEDURES

A PROCEDURE in Algol is a very general and flexible structure which includes as a subset the more or less generally recognized classes of subroutines and functions. It is generally used to specify a section of program, which usually represents an algoithm, as a somewhat independent piece of the program that can be called or reused many times from other parts of the program. This structure enables one to test various independent pieces of a program before putting them together in a more complicated way.

The procedure may be classified or subdivided into three categories, namely: normal Algol procedures (simply called procedures), library procedures (Chapter VII), and external procedures. First, let us consider the class of normal procedures.

10.1 INTERNAL PROCEDURES

These procedures represent a special type of block called a procedure block. Procedure blocks are defined by means of a procedure declaration and exhibit most of the normal properties of blocks.

10.1.1 THE PROCEDURE BLOCK

A procedure block consists of a procedure heading followed by either a standard block or a statement, which represents the body of the procedure. The procedure heading consists of three parts, namely: the procedure declaration with formal parameter list, the VALUE part, and the specification part.

10.1.1.1 THE PROCEDURE DECLARATION

The format of the procedure declaration is as follows:

FORMAT:

< type > PROCEDURE α ;

< type > PROCEDURE α (F_1, F_2, ..., F_k) ;

< type > PROCEDURE α (F_1)L_1:(F_2)L_2:(...)L_{k-1}:(F_k) ;

where α is the name of the procedure, F_i are the names of the formal parameters of the procedure, L_i (if present) are letter strings which represent parenthetical comments, and < types > (c.f. Chapter V) is one of the following: REAL, REAL 2, INTEGER, BOOLEAN, COMPLEX or STRING. Further it should be noted that < type > may be empty. However, if < type > is empty, the procedure cannot be used in the functional sense (see below).

When the procedure declaration is encountered at run-time, space is created for each of the formal parameters in a fashion similar to that obtained for local variables in a block. The formal parameters are considered as local identifiers to the defining procedure block, and may be used in the global sense in any block nested inside the procedure block. When the procedure is called, space is allotted for all local variables and parameters, and all of the actual parameters are moved into the cells allotted to the formal parameters.

The procedure declaration must be present in all procedure blocks and is the first part of the procedure heading.

EXAMPLES:

```
PROCEDURE INVERT (A, B) ;
REAL PROCEDURE SIMPS (A,B) FUNCTION:
      (F) ERROR: (DELTA) ;
```

10.1.1.2 THE VALUE PART

This part of the procedure heading is optional and has the following format:

FORMAT:

VALUE < formal parameter list > ;

The < formal parameter list > consists of those formal parameter identifiers contained in the procedure declaration, which are to be considered as values, separated by commas. These parameter values are obtained from the procedure call when the procedure block is entered and remain fixed (unless altered by a replacement statement) throughout the body of the procedure. Any formal parameter occurring in the VALUE part is considered to be a call-by-value parameter or simply a value parameter. A value parameter behaves identically to a local variable except that its initial value is obtained from the procedure call.

It should be noted that a value parameter which is an array identifier will cause the entire array supplied by the procedure call to be copied locally within the procedure block. This, in effect, may cause large amounts of memory space to be unexpectedly used when the procedure is called. Also, value parameters that correspond to labels will cause any switch variable or designational expression in the call to be evaluated upon entry, as expected.

If the arithmetic type of the actual parameters in the procedure call differ from those of the formal parameters, then an appropriate type conversion will be performed (if possible) for those cases in which the formal parameter is a value parameter. All other cases will result in an error message at run-time.

The value part (if present) must follow the procedure declaration and precede the specification part.

EXAMPLE:

```
VALUE X,A,Z;
```

10.1.1.3 THE SPECIFICATION PART

All formal parameters defined by a procedure declaration must be specified with regards to < type > in the specification

part of a procedure heading. The format of the specification part is as follows:

FORMAT:

⟨specification⟩ ⟨formal parameter list⟩ ;

The ⟨specification⟩ has one of the following forms:

⟨type⟩
ARRAY
⟨type⟩ ARRAY
PROCEDURE
⟨type⟩ PROCEDURE
LABEL
SWITCH
FORMAT

and ⟨type⟩ is one of the following:

INTEGER
REAL
REAL2
BOOLEAN
COMPLEX
STRING

The ⟨formal parameter list⟩ again consists of the formal parameter identifiers contained in the procedure declaration separated by commas.

The reason that all formal parameters must be specified is that the compiler must know the type of all parameters in order to compile proper machine code.

EXAMPLES:

```
INTEGER I,K;
REAL X,Y;
REAL ARRAY Z;
BOOLEAN PROCEDURE F;
STRING S;
```

The details of constructing a procedure block can be best described by analyzing several examples.

EXAMPLES:

```
PROCEDURE SPUR(A) ORDER:(N)RESULT:(S) ; VALUE N ;
ARRAY A ; INTEGER N ; REAL S ;
BEGIN INTEGER K ;
S: = O ;
FOR K:= 1 STEP 1 UNTIL N DO S:= S+A(K,K)
END
```

```
PROCEDURE TRANSPOSE(A)ORDER:(N) ; VALUE N ;
ARRAY A ; INTEGER N ;
BEGIN REAL W ; INTEGER I, K ;
FOR I:= 1 STEP 1 UNTIL N DO
      FOR K:+ I+1 STEP 1 UNTIL N DO
      BEGIN W:= A(I,K) ;
                A(I,K):= A(K,I) ;
                A(K,I):= W
      END
END TRANSPOSE
```

```
INTEGER PROCEDURE STEPS(U) ; REAL U ;
STEPS:= IF O <= U & U <= 1THEN 1 ELSE O
```

```
PROCEDURE ABSMAX(A) SIZE: (N,M)RESULT: (Y)SUBSCRIPTS:(I,K) ;
COMMENT THE ABSOLUTE GREATEST ELEMENT OF THE MATRIX A,
        OF SIZE N BY M IS TRANSFERRED TO Y, AND THE
        SUBSCRIPTS OF THIS ELEMENT TO I AND K ;
ARRAY A ; INTEGER N,M,I,K ; REAL Y ;
BEGIN INTEGER P,Q ;
Y:= O ;
FOR P:= 1 STEP 1 UNTIL N DO FOR Q:= 1 STEP 1 UNTIL M DO
IF ABS(A(P,Q))>Y THEN BEGIN Y:= ABS(A(P,Q)) ;I:=p;K:=Q
END END ABSMAX
```

```
PROCEDURE INNERPRODUCT (A,B)ORDER:(K,P)RESULT:(y);
VALUE(K);
INTEGER K,P; REAL Y,A,B;
BEGIN REAL S;
S:= O;
FOR P:= 1 STEP 1 UNTIL K DO S:= S+A*B;
Y:=S
END INNERPRODUCT
```

10.1.2 VALUE AND NAME PARAMETERS

All formal parameters that have been listed in a VALUE part are called value parameters. These parameters will be assigned a value corresponding to the value of the actual parameter in the procedure call upon entering the procedure block. Any changes made in the value parameters within the body of the procedure block will have no effect outside the body of the procedure.

Any formal parameters that have not been listed in a VALUE part are called name parameters. Upon entering the procedure block, these parameters will be assigned an address of a subroutine which calculates the address of the actual parameter within the body of the procedure block will be carried outside to the actual parameters supplied by the call in the calling block.

This property of name parameters permits a large number of results to be supplied to the calling program from within the procedure. One word of caution: This property can sometimes cause far-reaching and disastrous effects in the overall program by altering either the values of variables in the calling block or the catual parameters in the call when least expected.

10.1.3 FUNCTIONAL PROCEDURES

Procedures which are to be used in the functional sense (e.g., SIN, EXP) must have a < type > associated with the

procedure identifier (i.e., procedure name). The <type> declaration must be the first symbol of the procedure declaration. Also for the functional procedure to have a value associated with it, the procedure identifier must occur at least once as the left part of an assignment statement in the procedure body. In addition, at least one of these assignment statements must be executed on a given procedure call for a value to be assigned to the procedure. If more than one such assignment statement is executed within the body, then the last one executed before exiting from the procedure deterines the value associated with the procedure. Any other occurrences of the procedure identifier within the body of the procedure will be considered as calls on the procedure.

Caution: Restrictions on STRING procedures.

1. The value of the functional procedure is not copied (as in the case of string replacement), only a pointer to the value is kept. Consequently if the source of the value is a string variable whose value is changed before the exit from the procedure, the value of the functional procedure is changed accordingly.

   EXAMPLE:

```
STRING PROCEDURE SP;
   BEGIN STRING S(5);
      S:= 'ABCDE';
      SP:= S;
      S$(3,2):= 'XY'
   END OF SP
```

   The value of SP is 'ABXYE'.

2. It is meaningless to do a partial string replacement on the procedure name since it has no defined length. Its length is the length of the value. For the above example, the following statement is meaningless and not allowed:

$$SP\$(\epsilon_1,\epsilon_2) := S\epsilon$$

3. As a general practice, the functional value should be assigned to the procedure name in the outermost block of the procedure and not in any nested block. A failure to observe this practice may cause the value to be lost under certain circumstance without any error indication.

EXAMPLE:

```
REAL PROCEDURE FACTORIAL (N) ;
VALUE N ;
INTEGER N ;
BEGIN REAL S ;
INTEGER I ;
S:= 1.0
FOR I:= 1 STEP 1 UNTIL N DO S:= S*I ;
FACTORIAL:= S
END FACTORIAL
```

10.1.4 THE PROCEDURE CALL

A procedure call has the following format:

FORM:

$$\alpha(F_1, F_2, \ldots, F_k)$$

where α is the name of the called procedure and F_i are the actual parameters supplied to the procedure.

The correspondence between the actual parameters in the procedure call and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure call must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order. It is the responsiblity of the programmer to match the types and kinds of parameters in the lists. Failure to do so will lead to a runtime error except in the case of value parameters which will be converted if possible.

A formal parameter which occurs as the left part variable of an assignment statement within the procedure body and which is not called by value (see above) can only correspond to an actual parameter which is a variable. Also, a formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition, if the formal array parameter is called by value, the local array created during the call will have the same subscript bounds as the actual array. Correspondingly, a formal parameter which is used within the procedure body as a string identifier can only correspond to an actual parameter which is a string identifier unless the formal parameter is called by value. In this case, appropriate conversions of type will be made if possible. However, in the case of string arrays, no conversion of type is permitted.

10.1.5 COPY RULE

The procedure call acts as if the call were replaced by the statements in the procedure body with the following changes being made in the body.

1. All formal parameters quoted in the value part of the procedure declaration are assigned values of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body.

2. Any formal parameter <u>not</u> quoted in the value list is replace, throughout the procedure body by the corresponding actual parameter, after enclosing the catual parameter in parentheses whenever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local idnetifiers involved.

3. Finally, the modified procedure body is executed in place of the procedure call. Any global identifiers referenced from within the body of the procedure will be considered to be those in force at the point of the procedure declaration.

10.1.6 RECURSIVE PROCEDURE CALLS

The usual context for a recursive procedure call is an explicitly stated procedure call from within the body of the procedure itself. As an example, consider the following recursive form for the previous factorial procedure:

EXAMPLE:

```
REAL PROCEDURE FACTORIAL(N);
VALUE N ;
INTEGER N ;
FACTORIAL:= IF N=0 THEN 1.0 ELSE
        FACTORIAL (N-1)*N;
```

Another more obscure method for a procedure to be called recursively is to pass a call for the procedure into the procedure as a name parameter. As an example, consider the recursive procedure for integration using Simpson's Rule (Frank Olynyk, Comm. of ACM, Vol.7, June 1964, P.348).

EXAMPLE:

```
REAL PROCEDURE SIMPS (X,X1,X2,DELTA,F);
VALUE X1,X2, DELTA;
REAL X,X1,X2, DELTA,F;
BEGIN
    BOOLEAN TURING; REAL Z1,Z2,X3,H,K;
    TURING:= FALSE;
    IF X1 = X2 THEN BEGIN SIMPS:=0, GO TO BOX3 END;
    IF X1 > X2 THEN BEGIN H:=X1; X1:= X2;
            X2:=H; TURING:= TRUE END;
    X:= X1,Z1:=F; X:X2; Z3:= Z1:= Z1+F;
    K:= X2-X1

BOX:
    Z2:= 0;H:=K/2;
    FOR X:= X1+H STEP K UNTIL X2 DO Z2:= Z2+F
    Z1:= Z1+4*Z2;
```

```
        IF H*ABS((Z1-2*Z3)/(IF Z1 = 0 THEN 1 ELSE Z1))
             < DELTA THEN GO TO BOX2 ELSE Z3:= Z1;
        Z1:= Z1-2*Z2;
        K:=H; GO TO BOX;

    BOX 2:

        IF TURING THEN H:= -H;
        SIMPS:= H*Z1/3;
    BOX 3:
    END SIMPS
```

Using this procedure, the double integral

$$\int_0^1 \int_0^{(1-X^2)^{1/2}} g(x,y)\ dy\ dx$$

would be evaluated by the recursive call:

```
DOUBLE:= SIMPS(X,0,1, DELTA, SIMPS(Y,0,SQRT(L-X**2),
         DELTA2, G(X,Y)))
```

10.1.7 GENERAL PROBLEM SOLVER

Now for those of the readers who feel that they have mastered the concept of procedures in Algol and who consider themselves sophisticated programmers, consider the following procedure called GPS for General Problem Solver (D.E. Knuth, Comm. of ACM. Vol.4, June 1961, P.271):

EXAMPLE:

```
REAL PROCEDURE GPS(I,N,Z,V);
REAL I,N,Z,V;
BEGIN
     FOR I:= 1 STEP 1 UNTIL N DO Z:= V;
     GPS:= 1
END GPS
```

Is not that the most harmless looking procedure you ever saw. Wait a minute, there is a lot of danger lurking

in the call-by-name parameters.

If we wish to calculate the innerproduct of the N-element vectors A and B, we simply write:

```
Z:=0; I:= GPS(I,N,Z,Z+A(I) * B(I))
```

But we can do much better than that. Suppose we want to multiply the array A(1:m, 1:n) by B(1:n, 1:p) and store the result in C(1:m, 1:p). This can also be done using GPS by writing

```
I:= GPS(I,1.0,C(1,1),0.0)*GPS(I,(m-1) *GPS(J,(p-1)*GPS
    (K,n,C(I,J),C(I,J)+A(I,K)*B(K,J)),C(I,J+1),0.0),
    C(I+1,1), 0.0);
```

You can also do problems which are quite unrelated to matrix multiplication with GPS. For example, the following will set P equal to the mth prime number:

```
I:= GPS(I,IF I=0 THEN -1.0 ELSE 1,P,IF 1=1 THEN 1.0
    ELSE IF GPS(A,I,Z, IF A = 1 THEN 1.0 ELSE
    IF ENTIER (A) * (ENTIER(I)//ENTIER(A)) = ENTIER(I)
    AND A<I
    THEN 0.0 ELSE Z) = Z THEN
    (IF P<M THEN P+1 ELSE P * GPS(A, 1.0, I, -GPS(A,
    1.0,P,I))) ELSE P)
```

In fact, using GPS we can actually compute any computable function using a single ALGOL assignment statement.

10.2 EXTERNAL PROCEDURES

External procedures in ALGOL can be coded in any of two ways, namely:

1) Another ALGOL procedure compiled separately.
2) A procedure coded in METASYMBOL following specified rules.

The coding rules vary depending upon the choice of coding methods. Each method in turn also specifies the form of the procedure declaration within the main ALGOL program.

10.2.1. PROCEDURE DECLARATION

If the external procedure has been coded in ALGOL and compiled separately, then the name of the procedure must be defined in the amin program by means of an EXTERNAL procedure declaration. The form of the declaration becomes in this case:

EXTERNAL < type > PROCEDURE $\alpha_1, \alpha_2, \ldots, \alpha_n$;

where < type > specifies the type of result associated with the procedure in the usual sense, and $\alpha_1, \alpha_2, \ldots, \alpha_n$ are the names of the desired external procedures. The omission of the type specification indicates that the procedure cannot be used in the functional sense. Also, a procedure defined in this manner will always be called in the recursive sense by the main program. An external procedure of this form can also be written as a METASYMBOL program, but the coding details are so complicated that they will be omitted from this note. For further details, see the internal documentation manual for the ALGOL compiler.

If the external procedure is to be coded in METASYMBOL, then the easiest method of handling the calling sequence is to define the procedure as a METASYMBOL procedure. This means that the procedure cannot call itself from within the METASYMBOL coding, and all expressions passed into the procedure will be evaluated before entry to the procedure. The form of the external procedure declaration within the main program becomes:

EXTERNAL METASYMBOL < type > PROCEDURE $\alpha_1, \alpha_2, \ldots, \alpha_n$;

where < type > and α_1 through α_n have the same meanings as the recursive case. Procedures of this form cannot be passed into other procedures as parameters.

10.2.2 PROCEDURE CALLS

The form of the procedure call for external procedures is the same as that for normal internally defined or library procedures, namely:

$$\alpha(F_1, F_2, \ldots, F_n)$$

where α is the name of the external procedure and F_1 through F_n are the actual parameters supplied to the procedure (Expressions).

In the case of the external METASYMBOL procedures the calling sequence for the above procedure call is:

```
                BAL,BO          *α'
                DATA,1          REG,AR0,0,n
                G               T1,AR1,L1,ARRESSSF1
                .
                :
                G               Tn,ARn,Ln,ADDRESSFn
G               COM             8,4,3,17
BO              EQU             0
B2              EQU             2
```

where α' is an address in OWN storage which has the form:

α' DATA α

T_1 denotes the kind of actual parameter as follows:

T_i (in Hex)	Meaning
00	Simple expression or variable
10	An ARRAY name
2j	Denotes a procedure name
	j=0, an Algol-60 procedure
	j=1, a recursive library procedure
	j=2, an external METASYMBOL procedure

3j	Denotes a label j=0, a simple label j=1, a SWITCH label j=2, a FORMAT label

AR_i denotes the type of arithmetic that the actual parameter has, as described below:

AR_i	Meaning
0	Undefined or universal
1	Boolean
2	String
3	Integer
4	Real
5	Real 2
6	Complex

AR_0 is the arithmetic of the result of a functional procedure, and finally L_i denotes the form of the actual parameter as follows:

$ADDRESS_{F_i}$

L_i	Meaning	Interpretation
1	Identifier	relative to B2
2	Constant	actual
3	Result	relative to B2
4	Indirect result	indirect relative to B2
5		
6	OWN variable	actual

The addresses of the actual parameters must be interpreted properly as indicated above. The result of the procedure, if it is being used in the functional sense, is left in register REG which is a double word address. If the result is a STRING, INTEGER, or BOOLEAN, it should be left in the odd word of the double word register. REAL results are left in the even word while COMPLEX and REAL 2 results occupy both words. See the internal documentation manual for the proper format of the STRING descriptor. All procedure, switch, and label identifiers have their actual addresses stored in OWN storage. See the Internal Documentation Manual on how to reference these quantities.

All run-time checking is the problem of the externally coded METASYMBOL program. The first line of the METASYMBOL program must be a DEF of the procedure name. The exit address is contained in Register 0 as indicated. The proper processing of the linkages is the problem of the METASYMBOL program along with proper return linkages.

10.2.3 OPERATIONAL TECHNIQUES

When compiling an external ALGOL procedure, the following rules should be observed.

1. The first statement must be a standard procedure declaration.
2. Each external procedure must be compiled separately.

When writing an external METASYMBOL procedure, the following rules should be observed.

1. All registers should be saved upon entry and restored upon exit. The link register, Register 0, must be adjusted accordingly upon exit.
2. An actual parameter cannot be a procedure name, a non-typed procedure call, a switch designator, or a designational expression of the form IF-THEN-ELSE.

This chapter deals with the means of expressing the 'flow of control' of an algorithm which has been described in c compiler language. The order of execution of statements is as important to the description of an algoithm as are the statements themselves.

Control Statements are divided into three sub-classes:

Unconditional control statements which transfer control to other parts of the program (the GO TO statement).

Conditional control statements which execute statements contingent on given criteria (the IF statement).

Iterative control statements which execute statements repetitively (the FOR statement).

6.1 THE GO TO STATEMENT

The GO TO statement provides the ability to transfer control from one part of the compiled program to another.

GENERAL FORM:

GO TO ε or GO ε

where ε is a statement label, switch variable or designational expression.

The statement with the label specified by ε will be executed immediately after the GO TO statement.

EXAMPLES:

GO TO START

GO S(K)

GO TO IF X > Y THEN POGO ELSE 5(3+1)

Note: In the second example, if K is within the range of definition of the switch S then transfer of control is made to the indicated statement label. However, if the subscript expression of a switch is outside the range of definition of the switch, the GO TO statement is equivalent to a dummy statement.

6.2 THE IF STATEMENT

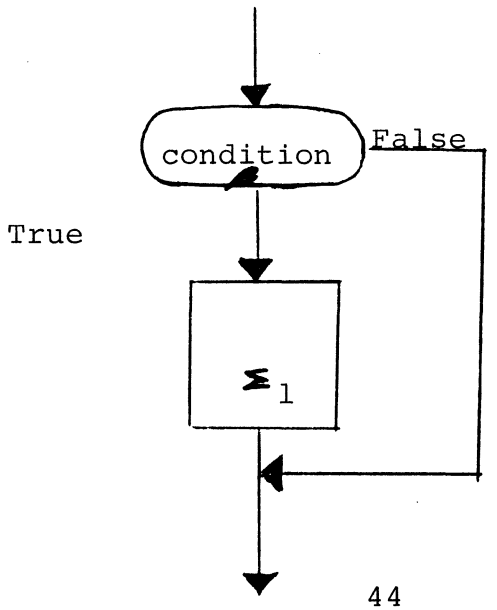
The IF statement provides the means of indicating that the next statement in sequence is to conditionally executed.

FIRST GENERAL FORM:

IF β THEN Σ_1

where β is a Boolean expression -- a 'condition' - and Σ_1 is any statement.

The action of the first form of the IF statement is described graphically by means of the following flowchart:



If the Boolean expression β is true, the statement Σ_1 is executed; if β is false, Σ_1 is skipped over. In either case, control continues in sequence with the next statement.

EXAMPLES:

```
IF X*2 > 7 THEN GO TO HOME
IF (1 = J) THEN A(1,J) := 1
IF (M = 0) | (N¬=0) THEN GO TO LAST
IF (P EQIV R) | (P EQIV S) THEN K:= B(J)
IF (X<=0) & FLAG THEN X := ABS(X)
IF U| V & (X<2.4) THEN BEGIN U:= 0;
    V:= 0; GO TO REPEAT END
```

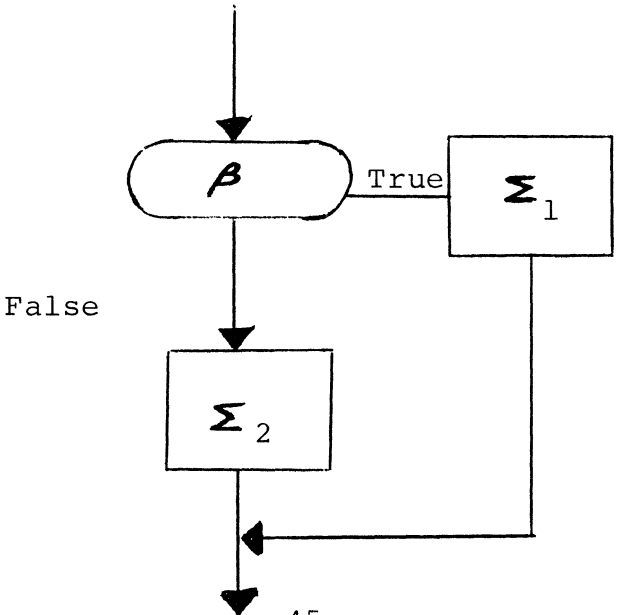
A second form of the IF statement provides for a choice of alternatives based on the rsult of a condition.

SECOND GENERAL FORM:

$$\underline{\text{IF}}\ \beta\ \underline{\text{THEN}}\ \Sigma_1\ \underline{\text{ELSE}}\ \Sigma_2$$

The interpretation of this statement is that if β is true then the statement Σ_1 is performed and Σ_2 is omitted. On the other hand, if β is false then the statement Σ_2 is performed while omitting Σ_1.

The following flow chart will serve to explain this statement more precisely.



EXAMPLES:

```
COMMENT EVALUATED THE LEGENDRE POLYNOMIAL OF
DEGREE N IN X USING A RECURSION RELATION

        REFERENCE: FOURIER SERIES AND BOUNDARY
                   VALUE PROBLEMS,
                   R.V. CHURCHILL
                   MC-GRAW HILL, 1941, Page 180;

IF N<O THEN BEGIN COMMENT YOU MADE AN ERROR
                    PN:= 1.0@38 END ELSE
IF N = O THEN PN:= 1 ELSE
IF N = 1 THEN PN:= X ELSE
BEGIN PC:= X; PR:= 1
   FOR M:= STEP 1 UNTIL N DO BEGIN
PN:= (X*(2*M-1)*PC-(M-1)*PR)/M;
PN:= PC; PC:= PN END END;
NTHDEGREELEGENDREPOLYNOMIALINX:= PN;
```

6.3 THE FOR STATEMENT

The FOR statement finds its principal use in the control of an iteration where the statement or statement group to be interated involves a variable (the iteration variable) which must take on a succession of values. It is also used to cause a statement to be executed a predetermined number of times.

GENERAL FORM:

FOR Γ:= <for list> DO Σ

where Γ is a variable, < for list > is a list of "for list elements" defined below, and Σ is any statement, compound or not.

The "for" list describes the sequence of values that the variable τ is to assume. The statement Σ will be executed for each of these values. After the iteration list has been exhausted, the statement following Σ will be executed.

There are three forms of "for list elements" which can be used to construct a "for" list. Each of the "for list elements" are separated from each other by commas. A summary of the possible forms is given below.

FIRST FORM:

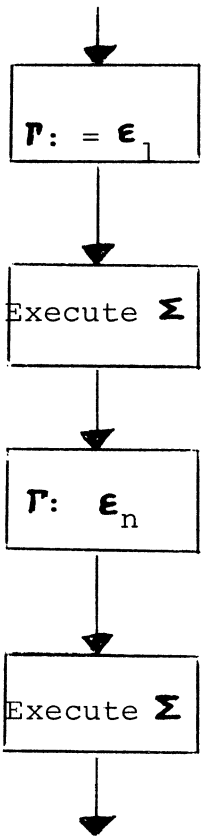
$$\epsilon$$

where ϵ is an arithmetic expression. The iteration variable τ will assume the value and the statement Σ will be executed before going on the next "for list element".

A FOR statement consisting of only these types of elements becomes:

$$\text{FOR } \tau := \epsilon_1, \epsilon_2, , \epsilon_3, \ldots\ldots, \epsilon_n \text{ DO } \Sigma$$

which can be interpreted by a flow chart as:



That is, the variable τ is successively given the values of ε_1, ε_2, and so on through ε_n. The statement Σ is executed once for each value which τ assumes.

EXAMPLES:

```
FOR PRIME:= 2,3,5,7,11,13,17 DO A(1):= PRIME
FOR X:= 0,Y+3*Z,IF Z > Y THEN Z ELSE Y, MAX(Z,Y)
                                    DO WRITE (X)
```

SECOND FORM:

$$\varepsilon_i \text{ STEP } \varepsilon_d \text{ UNTIL } \varepsilon_f$$

where ε_i, ε_d, and ε_f are arithmetic expressions.

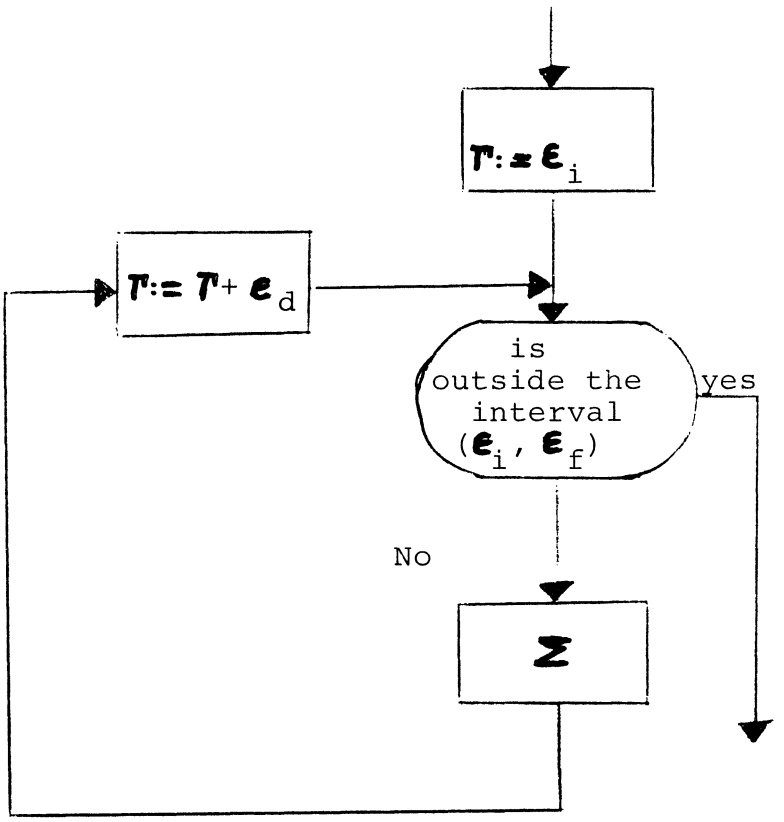
A FOR statement consisting of only one of these elements takes on the form:

$$\text{FOR } \tau := \varepsilon_i \text{ STEP } \varepsilon_d \text{ UNTIL } \varepsilon_f \text{ DO } \Sigma$$

This form is equivalent to the simpler statements:

$$\tau := \varepsilon_i;$$
$$S : \text{IF } (\tau - \varepsilon_f) * \text{SIGN } (\varepsilon_d) <= 0 \text{ THEN}$$
$$\text{BEGIN } \Sigma; \tau := \tau + \varepsilon_d; \text{ GO TO } S \text{ END}$$

which is described graphically by the following flow chart:



In order to make this more clear, let us examine the case where ϵ_d is positive and negative, separately. If ϵ_d is positive the form reduces to

$\tau := \epsilon_i$;

S : IF $(\tau - \epsilon_f) <= 0$ THEN

BEGIN Σ; $\tau := \tau + \epsilon_d$; GO TO S END

If ϵ is negative

$\tau := \epsilon_i$;

S : IF $(\epsilon_f - \tau) <= 0$ THEN

BEGIN Σ; $\tau := \tau + \epsilon_d$; GO TO S END;

Note that in all cases if the test fails initially, the triplet is considered vacuous, and the statement Σ will not be executed at all.

On exit from the FOR statement, the value of the iteration variable τ is considered to be undefined unless outside transfer is effected.

EXAMPLES:

```
COMMENT EVALUATE INNER PRODUCT OF U ( ) AND V ( );
   DOT:= 0; FOR I:= 1 STEP - UNTIL N DO
   DOT:= DOT + U(I)*V(I);

COMMENT LINEAR EQUATION SOLVER FOR A(,)*X( )=A(,N+1),
            THE ANSWERS ARE PUT INTO A(,N+1);

FOR J:=2 STEP 1 UNTIL N+1 DO FOR I: 1 STEP 1 UNTIL N DO
         BEGIN SUM:=0; FOR K:= 1 STEP 1 UNTIL MIN
                                    (I-1,J-1) DO
                        SUM:= SUM + A(I,K)*A(K,J);
       A(I,J):=A(I,J)-SUM; IF I<J THEN A(I,J):=
                                    A(I,J)/A(I,I) END;

FOR I:= N-1 STEP - I UNTIL I DO
        BEGIN SUM:= 0; FOR K:=N STEP -1 UNTIL I+I DO

             SUM:=SUM + A(I,K)* A(K,N+1);
       A(I,N+1):= A(I,N+1) - SUM END;
```

The third and final form that a "for list element" may assume is called a WHILE element.

THIRD FORM:

$$\varepsilon \text{ WHILE } \beta$$

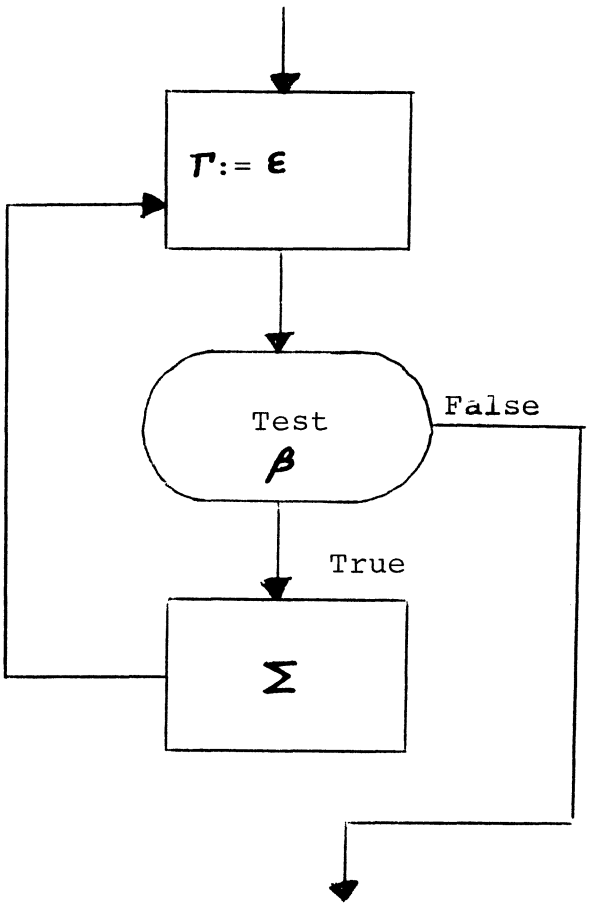
where ε is an arbitrary arithmetic expressions and β is a Boolean expression. A FOR statement having only one WHILE element would have the form;

$$\text{FOR } \tau := \varepsilon \text{ WHILE } \beta \text{ DO } \Sigma$$

The interpretation of this statement is that first τ will be set equal to ϵ. Then if β is true statement Σ will be executed. After the execution of Σ, τ will be replaced by ϵ and again β will be tested. If on the other hand, β is false then the statement Σ will be skipped and control will resume with the statement following the FOR statement. To make the interpretation of the construction clearer, the FOR statement can be replaced by the following set of single statements.

L : τ := ϵ;
IF β THEN BEGIN Σ; GO TO L END;

which can be expressed by the following chart:



The value of the iterated variable r in the FOR statement upon exit will in general be undefined.

6.3.1 JUMPS IN AND OUT OF FOR STATEMENTS

A GO TO into a FOR statement leads to undefined operation. A GO TO out of a FOR statement is legal and the iterated variable retains its current value.

EXAMPLE:

```
COMMENT INITIALIZE VECTOR V TO ZERO; I:=0;
FOR I:= I+I WHILE I <= N DO V(I) :=0;
```

It should of course be realized that all three types of for list elements can be combined together in any order to form a general ‹for list› to be used within a FOR statement. The sequence of values which results is the expected one, it should be noted, however, that e.g.

> FOR J:= 1 STEP 1 until 6, J DO 7 and not 6 is the value assigned to J the last time round.

Full computational procedures for certain standard functions of mathematics are defined in Algol. These functions are called by means of appropriate identifiers. Except for MOD, the arguments of the function are enclosed in parentheses immediately following the function's name. The form of the call is:

$$\text{NAME } (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)$$

where NAME is the identifier of the function and $\varepsilon_1, \varepsilon_2, \ldots. \varepsilon_n$ are the desired expressions for the input arguments to the function. The set of standard functions that are available for use with Algol are divided into three classes: intrinsic functions, type transfer functions, and library functions.

7.1 INTRINSIC FUNCTIONS

There is a small group of functions called intrinsic functions, the definitions of which are a part of the compiler. Each of the intrinsic functions is discussed in turn.

ε_1 MOD ε_2

The function MOD requires two integer arguments. The value of the function is an integer evaluated as

$$\varepsilon_1 - \varepsilon_2 * (\varepsilon_1 / \varepsilon_2)$$

where the brackets indicate integer part; i.e., the sign is the same as ε_1.

SIGN (ϵ)

The function SIGN has a single argument. If this argument is positive, the result will be +1; if zero, the result is zero; if negative, the result is -1. The result of the SIGN function will be of integer type.

LENGTH (ϵ)

The function LENGTH requires a single string expression as an argument. The value of the function is an integer denoting the length of the string.

ABS(ϵ)

The function ABS has a single argument. The result of the ABS function will be the absolute value of the argument and will be of the same type as its argument except in the case of a complex argument which has a real result.

RE(ϵ)

The function RE requires a single complex expression as an argument. The value of the function is a real number denoting the real part of the argument.

IM(ϵ)

The function IM requires a single complex expression as an argument. The value of the function is a real number denoting the imaginary part of the argument.

CONJ(ϵ)

The function CONJ requires a single complex expression as an argument. The value of the function is a complex number denoting the complex conjugate of the argument.

CMPLX (ϵ_1, ϵ_2)

The function CMPLX requires two real expressions as arguments. The value of the function is the complex number $\epsilon_1 + i\,\epsilon_2$.

7.2 TYPE TRANSFER FUNCTIONS

A set of type transfer functions are provided in the compiler to allow quantitites of type real or integer to be treated as boolean without altering their internal system representation, and vice versa.

In the following, let represent an arithmetic expression and represent a boolean expression.

REAL ( )

The function REAL yields a value of type REAL. This allows arithmetic operations to be performed on BOOLEAN quantities.

$$\text{REAL (TRUE)} = -(1-2^{-24}) \times 16^{63}, \text{ i.e. X'FFFFFFFF'}$$

$$\text{REAL (FALSE)} = 0$$

INTEGER ( )

The function INTEGER yields a value of type INTEGER. This allows arithmetic operations to be performed on BOOLEAN quantities.

$$\text{INTEGER (TRUE)} = -1$$
$$\text{INTEGER (FALSE)} = 0$$

BOOLEAN ( )

The function BOOLEAN yields a value of type BOOLEAN. This value, as a logical value, is considered to be TRUE if $\neq 0$, and FALSE if $=0$. This allows Boolean operations to be carried out on arithmetic quantities of type INTEGER or REAL.

Used in conj-nction, these functions provide a facility for handling masking operations since the logical operators (Chapter III) operate on the entire word in the system.

7.3 LIBRARY FUNCTIONS

The set of library procedures are divided into three classes:

a) Input/Output Procedures
b) Standard Mathematical Procedures
c) Recursive Procedures

7.3.1 INPUT/OUTPUT PROCEDURES

The set of "Input/Output" library procedures are summarized below with a brief description. They are fully described in Chapter VIII.

NAME	DESCRIPTION
READ $(X_1,X_2,\ldots,X_k)$	This is the input procedure.
WRITE$(X_1,X_2,\ldots,X_k)$	This is the output procedure.
OPENREAD (X_1, X_2)	This procedure is used to open the input file.
OPENWRITE (X_1, X_2)	This procedure is used to open the output file.
CLOSEREAD	This procedure closes the input file.
CLOSEWRITE	This procedure closes the output file.
INPUT$(X_1, X_2,\ldots,X_n)$	This procedure moves data from the input buffer into the list of variables.
OUTPUT$(X_1, X_2, \ldots,X_n)$	This procedure moves the list of variables into the output buffer.
ERR(X_1)	This procedure is used to set the error return point for a read.
EOF(X_1)	This procedure is used to set the end-of-file return point for a read.
REWIND(X_1)	This procedure causes the designated unit to be rewound.

BACKSPACE (X_1)	This procedure causes the designated unit to be back-spaced one logical record.
ENDFILE(X_1)	This procedure causes an end-of-file mark to be written on the specified unit.

7.3.2 STANDARD MATHEMATICAL PROCEDURES

The standard mathematical procedures require only one input argument and the type of the result depends upon the type of the input argument. This relationship is described below:

TYPE OF INPUT	TYPE OF RESULT
INTEGER	REAL
REAL	REAL
REAL2	REAL 2
STRING	REAL
COMPLEX	COMPLEX

The mathematical functions that fall into this subclass are:

NAME	DESCRIPTION
ARCTAN(**E**)	inverse tangent
COS(**E**)	cosine
ENTIER(**E**)	greatest integer*
EXP(**E**)	exponential to base e
LN(**E**)	natural logorithm
SIN(**E**)	sine
SQRT(**E**)	square root

*The ENTIER function requires an input of type REAL or REAL 2 and the type of the result is INTEGER.

7.3.3 RECURSIVE PROCEDURES

Recursive library functions are those whose name may be passed as arguments to a procedure. The legal names are:

ABS	LENGTH
ARCTAN	LN
COS	SIGN
ENTIER	SIN
EXP	SQRT

The input arithmetic is the arithmetic of the argument while the output arithmetic is the arithmetic specified for the dummy function name in the procedure specification.

READ AND WRITE STATEMENTS
FORMAT DECLARATION
ERR AND EOF STATEMENTS
REWIND, BACKSPACE AND ENDFILE STATEMENTS

We now pursue in some detail the subject of communication with the computer, i.e., the details (with examples) of the card reader, printer, tape and disc 1/O (Input/Output) processes. The reader is warned that this chapter is difficult going. To make his life easier a substantial number of details have been omitted during the first encounter with some of the concepts -- the details are all in here -- but not necessarily where they belong according to strict logical rules. The reader is advised to read through the whole chapter before trying to "pin down" any details.

8.1 INPUT/OUTPUT

All input/output is performed by the RXDS FORTRAN IV 1/O package.

8.1.1 THE READ STATEMENT

The READ statement is broken down into three separate statements, namely:

1) OPENREAD (‹DCB number› , ‹format designator›)

where any of the above arguments may be omitted.

‹DCB number› specifies which user DCB is to be associated with the operation. If this argument, which may be an unsigned integer constant or an integer variable is omitted then "5", the normal card reader, will be assumed.

‹format designator› refers to the identifier BINARY or the name of a format specification list which appears in a FORMAT declaration previously encountered in the program. This list described the BCD input record's

format only in those cases where the user desires to make a rigid specification.

In the cases where the programmer does not wish to specify a format there is an implied format that will be used. The implied input format corresponds to the widthless numeric input format (8G). This means in the case of card input that the quantities called for in the INPUT statement are understood to be supplied on a card (or cards) in the o order specified with 8 values/card, and each number separated by either a blank or a comma. Complex values are processed as two separate items; the real and imaginary parts require individual specifications. When using the G format specification on string variables, the string values must start on a separate input record in the first character position. Succeeding values must start in new input records.

The effect of this statement is to open an input file from the indicated device in preparation for a subsequent INPUT statement.

2) INPUT (< actual input parameter list >)

where

<actual input parameter list> is a sequence of variables - simple, subscripted, or array identifiers - or a partial string operation on a string variable, separated by commas. The effect of this statement is to transmit values from the input media to the indicated list of variables. This statement may be the object of a FOR clause when inputing part of an array. If the variable is an array name, then the entire array is input such that the subscripts are varied from right to left.

3) CLOSEREAD

This statement closes the input file after an INPUT operation.

EXAMPLE 1 (Using the implied Format)

```
INTEGER A,B,C;
OPENREAD;
INPUT (A,B,C);
CLOSEREAD
```

This reads the values of three variables (A,B,C) from a card. The first number encountered is the value of A, the second is B, etc. until the list is satisfied. Any remaining values on the card are then lost completely.

EXAMPLE 2 (Using the implied Format)

```
INTEGER N,I,J;
REAL ARRAY A(I:IO,1:1O);
OPENREAD;
INPUT (N);
FOR I:= 1STEP 1 UNTIL N DO
FOR J:= 1STEP 1 UNTIL N DO
INPUT (A(I,J));
CLOSEREAD
```

The effect of this example is to read in a value of N and then to read in the array A a row at a time. Since the implied format is used, eight (8) values will be taken from each card.

If only simple variables, subscripted variables or entire arrays are to be initialized, the three statements can be compressed into one, namely:

READ (⟨CDB number⟩ , ⟨format designator⟩ , ⟨actual input parameter list⟩)

then Example 1 becomes

```
INTEGER A,B,C;
READ (A,B,C)
```

It should be noted that in this case a DCB number cannot be specified if the ⟨format designator⟩ is omitted due to an ambiguity in the notation.

If in Example 2, N is always 10, then the statements can be rewritten as

```
INTEGER I,J;
REAL ARRAY A(I:10,1:10);
READ (A)
```

8.1.2 THE WRITE STATEMENT

As in the READ statement, the WRITE statement is broken down into three separate statements, namely:

1) OPENWRITE (<DCB number> , <format designator>)

where any of the above arguments may be omitted.

<DCB number> specifies which user DCB is to be associated with the operation. If this argument, which may be an unsigned integer constant or an integer variable, is omitted then "6", the normal line printer, will be assumed.

<format designator> refers to the identifier BINARY or the name of a format specification list which appears in a FORMAT declaration previously encountered in the program. This list described the format of a BCD output record in those cases where the user does not wish to take advantage of the "implied format". If no format designator is used, the format is understood to be (8(1XG15.8)). Complex values are processed as two separate items- the real and imaginary parts require individual specifications. If the G format specification is for a string variable, then the string values will be outputed in separate records starting in the second character position.

The effect of this statement is to open an output file on the indicated device in preparation for a subsequent OUTPUT statement.

2) OUTPUT (<actual output parameter list>)

where

<actual output parameter list> is a sequence of expressions separated by commas. The effect of this statement is to transmit the values of the expressions to the output file. This statement may be the object of a FOR clause when outputing part of an array. If the expression is an array name, then the entire array is output such that the subscripts are varied from right to left.

3) CLOSEWRITE

This statement closes an output file after an OUTPUT operation.

EXAMPLE 1 (Using the implied Format)

```
INTEGER A,B,C;
OPENWRITE;
OUTPUT (A,B,C);
CLOSEWRITE
```

This will write out the values of A,B, and C, using the format (8(1XG15.8))

EXAMPLE 2 (Using the implied Format)

```
INTEGER I,J,N;
REAL ARRAY A(I:10,1:10);
OPENWRITE;
FOR I:= 1 STEP 1 UNTIL N DO
FOR J:= 1 STEP 1 UNTIL N DO
OUTPUT (A(I,J));
CLOSEWRITE
```

This example prints the array A a row at a time.

If only simple expressions or entire arrays are to be printed, then the three statements may be compressed into one namely:

WRITE (<DCB number> , <format designator> , <actual output parameter list>) and Example 1 may be rewritten as

```
INTEGER A,B,C;
WRITE (A,B,C)
```

Likewise, in this case a < DCB number> cannot be specified if the <format designator> is omitted.

If in Example 2, N is always 10, then it can be rewritten as

```
INTEGER I,J;
REAL ARRAY A(1:10,1:10);
WRITE (A)
```

8.1.3 BCD AND BINARY INPUT/OUT*

Input/output records may either be in BCD or binary. A file is declared to be binary by including as its <format designator> the reserved symbol BINARY in either the READ or WRITE statement- otherwise the file is BCD

8.1.3.1 BCD INPUT/OUTPUT

READ/WRITE statements with no specified < format designator> , or a <format designator> referencing a FORMAT declaration are used to process BCD records. The form of conversion to be performed between the internal data and the external fields is specified by a FORMAT declarations.

A formatted READ statement causes the character string in the external record to be converted, according to the FORMAT specified, into binary values, which are then assigned to the variables appearing in the input list. Converselu, a formatted WRITE statement converts internal values into character strings and outputs them.

Each formatted READ/WRITE statement begins processing with a new record. Thus, it is not possible to process a particular record using more than one READ or WRITE statement. More than one record may be processed by the statements if

* The following sections are taken with the XDS Sigma 5/7 Reference Manual with appropriate modifications.

if specifically requested by the FORMAT declarations. Attempting to read (or write) more characters on a record than are (or can be) physically present does not cause a new record to be begun; on output the extra characters are lost, on input they are treated as blanks.

A BCD record may have a maximum size of 132 characters. Certain devices may impose other restrictions on the size of records, e.g., a card contains 80 characters. A record may contain as few as zero characters, in which case it is considered to be blank (empty). Note, however, that on devices such as magnetic tape on disc, the FORMAT declaration may determine the actual size of record written. (See the RXDS Sigma FORTRAN IV Operations Manual for a complete description of BCD records.)

8.1.3.2 BINARY INPUT/OUTPUT

READ/WRITE statements with a BINARY <format designator> are used to process information in internal (binary) form and are designed to provide temporary file storage on magnetic tape and disc.

The binary READ/WRITE statements process data as a string of binary digits, arranged into words depending on the size of the items in the input/output list. All the items appearing in the input/output list of a binary READ/WRITE statement are contained in one logical record.

A logical record may consist of several physical records; however, as far as the programmer is concerned, it is treated as a single record. (See the RXDS Sigma FORTRAN IV Operations Manual for a description of the format of intermediate binary information.) This means that the information output by a single binary WRITE statement must be input by one and only one binary READ statement. It is permissible to read less information than is present in the record. If the input list requests more data from a binary record than is present, an error will occur. There is no limit to the number of items that can be processed by

a single binary READ/WRITE statement; only one logical record will be read or written regardless of the amount of data to be transferred.

The records produced by a binary WRITE do not consist of just the data to be generated. Control words are included in the records to facilitate reading or backspacing the proper number of physical records. Thus, the information produced by an intermediate binary WRITE statement is meant to be read subsequently by a binary READ statement.

8.2 THE FORMAT DECLARATION

The general form of the FORMAT declaration is given by

FORMAT ⟨format label⟩ (⟨format phrases⟩)

where

⟨format label⟩ is the name to be associated with the format.

⟨format phrases⟩ are the RXDS-FORTRAN phrases which describe the output on the printed page. These phrases, except for strings, are the same as those used by RXDS-FORTRAN and the user is referred to the RXDS-FORTRAN manual for a discussion of Formats. Appendix A is a reprint of the appropriate sections from the RXDS-FORTRAN manual for the users convenience. The only non-acceptable formats are the $ and H formats, i.e., only the ' format is recognized by the compiler.

In the case of strings, the only proper format is nA1, where n is an integer denoting the length of the string.

A warnin g message is given if the format specification is extra long, i.e., longer than 400 characters - 5 cards.

EXAMPLES:

```
FORMAT F1('INVERTED MATRIX IS'
FORMAT F2(3A1,2X,F15,8,5X,315
FORMAT F3('A=',F15.8,'B=',F15.8,
       'C=',F15.8,'ROOT=',F15.8)
```

8.3 THE ERR AND EOF STATEMENTS

Read statements may optionally include a specification of action to be performed if an error occurs or an end-of-file mark is read. The statements, which must be executed before a READ or INPUT statement, are written

ERR (< designational expression >)
EOF (< designational expression >)

where the value of the < designational expression > is the label in the program that control will be transferred to if during the processing of a READ statement an unrecoverable error occurs, or an end-of-file mark is encountered. When the transfer is effected, the value of the < designational expression > must be within the scope of the block containing the READ statement.

The return points remainnin effect for all subsequent READ statements. The original system defaults may be reinstated by omitting the argument <designational expression> from the call.

EXAMPLE:

(Setting up return points)

```
        BEGIN
        REAL A,B,C; FORMAT F1 ('CARD READER ERROR');
        ERR(L1); EOF(L2);
L3      READ(A,B,C);
        ANS = SQRT((B*B-4*A*C)/(2*A));
        WRITE(ANS); GO TO L3
L1      WRITE(F1);
L2      END OF PROGRAM
```

8.4 THE REWIND, BACKSPACE AND ENDFILE STATEMENTS

The following set of statements enables the programmer to manipulate magnetic tape and sequential disc files. In all cases, < DCB number > is an unsigned integer constant or integer variable whose value specifies the unit the operation is to be performed on.

8.4.1 REWIND STATEMENT

This statement is expressed as

REWIND (< DCB number >)

Execution of a REWIND statement causes the unit whose logical unit number is < DCB number > to be rewound.

8.4.2 BACKSPACE STATEMENT

This statement has the form

BACKSPACE (< DCB number >)

When a BACKSPACE statement is executed, the unit referenced by the integer value < DCB number> is backspaced one logical record. For bnnary tapes, a logical record may consist of more than one physical record. In this case, a logical record is interpreted as all the information output by one binary WRITE statement.

REWIND and BACKSPACE statements that are executed for tapes already positioned at "load point" have no effect.

8.4.3 ENDFILE STATEMENT

This statement causes an end-of-file mark to be written on the specified unit, and has the form:

ENDFILE (< DCB number>)

Sometimes it is desirable to take a program that has been written for output on magnetic tape and assign that logical unit number to some other device (such as a line printer). Since such programs oftern write and end-of-file and rewind their tapes at the end of the job, it is permissible to specify an ENDFILE or REWIND operation on any device. When the device is not a magnetic tape or sequential disc file, the statements have no effect. It is not permissible to BACKSPACE such devices.

8.5 STRING INPUT/OUTPUT

When using unformatted or G format I/O, each operation begins on a new record and continues for as many records (cards or lines) as is necessary to contain the string. The I/O data item must contain the same number of characters as the current length of the string.

In the case of formatted I/O, the format phrase must be of the form:

nAl

where n is an integer denoting the current length of the string. However, the nAl format phrases can be interspersed with T,X, or ' specifications.

BLOCKS

In Algol 60, a program consists of compound statements or one block which in turn may contain within it many sub-blocks nested to any depth. These sub-blocks serve to derine a structure within a program which facilitiates the construction of a program by permittting it to be built up of pieces - each of which may be relatively independent of each other. These pieces may in turn be tested separately before putting them together in the final program.

An upper bound of 256 different blocks per program is imposed by the Algol 60 compiler. This bound should be adequate for most programs. The blocks are numbered from 1 to 256 in the order in which they are read by the compiler.

An example of a program with a complex block structure is given in Figure 1.

BLOCK FORMAT

A block consists of two parts; namely, the heading and the body. Neither of these parts may be empty, and the block head must precede the body of the block.

The block head consists of all declarations needed within the block. This forces all identifiers, procedures, labels, formats, arrays, etc. to be defined before they are used.

The body of the block on the other hand contains all statements pertinent to the block which are not declarations. This means that all of the active statements such as replacement statements must be located in the body of the block.

As an example of a block, consider the following do-nothing block.

```
DONOTHING:  BEGIN
            INTEGER I;
            REAL ARRAY A(I:20);
            FOR I:= 1 STEP 1 UNTIL 20 DO
            A(I):= 0.0
            END DONOTHING;
```

In the above example, the declarations INTEGER 1 and REAL ARRAY A(I:20) constitute the head of the block while the FOR statement and replacement statement form the body of the block.

9.2 DEFINING A BLOCK

A block is defined by enclosing a section of a program consisting of a head and body within a set of BEGIN - END statement parentheses, as in the above example. This, in effect, means that the word BEGIN followed immediately by some form of declarations defines a new block which is terminated by the END that matches the defining BEGIN. The double use of the BEGIN - END parentheses should be noted by the user. A group of statements enclosed by a BEGIN - END pair will form a compound statement if the statement following the BEGIN is not a declaration while on the other hand if this statement is a declaration then a new block will be defined rather than a compound statement.

EXAMPLE:

```
BEGIN
COMMENT     THIS PROGRAM INVERTS A SQUARE MATRIX A BY
INTEGER     N                    STRAIGHT GAUSS ELIMINATION;
READ        (N);
BEGIN       REAL ARRAY A(1:N,1:N), V(1:N);
            INTEGER I, J, C;
```

```
        READ (A);
        FOR C:= 1 STEP 1 UNTIL N DO
        BEGIN FOR I:= 1 STEP 1 UNTIL N-1 DO
        V(I):= A(1, I+I)/A(1,1);
        V(N):= 1/A(1,1);
        V(N):= 1/A(1,1);
        FOR I:= 1 STEP 1 UNTIL N-1 DO
        BEGIN FOR J:= 1 STEP 1 UNTIL N-1 DO
        A(I,J):= A(I+1,J+1) - A(I+1,1)*V(J);
        A(I,N):= -A(I+1,1)*V(N)
        END; FOR J:= 1 STEP 1 UNTIL N DO
        A(I,J):= V(J)
        END; WRITE (A)

END
END
```

9.3 LOCAL AND GLOBAL IDENTIFIERS

All identifiers that are declared explicitly within a given block are said to be local to that given block. Any identifiers that are defined in an outer block to the given block (i.e., in any block that contains the given block) and are not redefined in the given block are said to be global to the given block. Also any identifiers that are defined within an inner block (i.e., any block contained in the given block) will have no meaning in the given block and cannot be referenced from the given block.

Since a label is an identifier, this last statement means that all blocks are entered through their heads and it is impossible within the language to enter the middle of a block. Also all variables that are defined in a block and hence local to the block will be bound to the block.

When a block is entered (through its head), space will be "created" or taken away from the remaining available space in memory and assigned to all normally defined local variables in the block including arrays. The initial values of these variables in the block including arrays. The initial values of these variables are considered to be undefined. When an exit of any form, whether by means of a GO TO statement or

normajly through the end of a block, is made from a block, all memory space that is assigned to local normal variable storage is returned to the available space part of memory and can be used by other blocks for local variable storage. At times it is desirable to have variables retain their values from block entry to block entry. Since normal variables may have their values redefined upon each entry and thrown away on each exit, a special class of variables called OWN variables are defined. This class has permanently allotted space in memory and will retain their values from entry to entry. However, it should be noted that the identifier rule still holds, and these variables although residing in memory cannot be addressed from outside the defining block.

To further illustrate the concept of local and global identifiers consider the block displayed in Figure 1. The variables I, J, K, X, Y, Z and labels L1, L2 and local identifiers in block 1. Only I, K, X, Y, Z, L1, L2 will be global to block 2, while J is redefined and local in block 2 along with L, M, U, V. In block 3, I, Y, Z, L1, L2 from block 1 are global along with J, L, U, V from block 2. K, M, N, W, X are local in block 3. Consider the statement, L3. The global variable Y will be replaced by the sum of the local variable X with the product of the global variables V,Z. The statement L4 in block 4 looks the same as L3 in block 3. However, the v ariable X in this case refers to the variable X in block 1 rather than the variable X in block 3; hence the effect of the statements will be different.

In block 5, the labeled statement L5 has an erroneous GO TO L4 after the word THEN. The label L4 is defined in block 4 and has no meaning within block 5 since the blocks are disjoint. However, the GO TO L1 is correct and will send control to the statement L1 in block 1. This in effect will cause the program to re-enter block 2.

```
BEGIN
INTEGER I,J,K;                                    (Block 1)
REAL X,Y,Z;

L1:BEGIN                    (Block 2)
                            REAL,J,L,M,U,V;
                            BEGIN
                                                  (Block 3)
                                                  INTEGER K,M,N;
                                                  REAL W,X;
                                                  L3: Y:=X*V*Z;
                            END;

                            BEGIN
                                                  (Block 4)
                                                  BOOLEAN B1,B2;
                                                  INTEGER P,Q;
                                                  REAL Z1, Z2
                                                  L4: Y:=X+V*Z;
                            END
END;

L2: BEGIN                   (Block 5)

                            INTEGER V,Z;
                            REAL I, M;
                            BOOLEAN B1
                            L5: IF B1 THEN GO TO L4 ELSE
                            GO TO L1;

END

END
```

3. All space requested by GETPAGE must be released before exiting. (See BPM Reference Manual).

4. See the internal documentation manual on how to jump to a label supplied as an actual parameter, or how to use a switch passed as an actual parameter.

5. Each external procedure must be compiled separately.

XI...

The Algol Compiler is a completely integrated processor in the Batch Processing Monitor (BPM) system on the XDS Sigma 5/9. To achieve speed, the Algol Compiler uses a special program loader which integrates all of the library routines with the object code.

11.1 PROCESSOR CALL

The format of the processor call card is as described in the BPM manual where the processor name is ALGOL.

11.2 OPTIONS

The following options* are available with the Algol processor:

OPTION	FUNCTION
LS	Output a listing of the source program to the device/file assigned to the M:LO DCB.
NLS	No source listing is provided.
LO	Output a listing of the object (generated machine language) program to the device/file assigned to the M:LO DCB.
NLO	No object listing is provided.
AD	All declared real variables will be double precision.
NAD	All declared real variables will be single precision.
GO	Output relocatable object (ROM) to GO file.
NGO	ROM is not output to GO file.

* Options are separated by commas with no intervening blanks.

OPTION	FUNCTION
FL	Output binary object program to GO file (this program may only be loaded and executed by ALOAD).
NFL	Binary object program is not output to GO file.
BO	Output ROM to device assigned to M:BO DCB.
NBO	ROM is not output to device assigned to M:BO.
DO	Output compiler diagnostic information to device/file assigned to the M:LO DCB.
NDO	No compiler diagnostics will be output.
DM	Causes additional diagnostics to be outputed to the device/file assigned to the M:LO DCB when run-time error occurs.
NDM	Suppresses the additional diagnostics.
BD	Delineates the start and end of each block.
NBD	No message given to delineate blocks.
SC	Algol code occupies columns 1-72 of the card.
NSC	Algol code occupies columns 1-80 of the card.
CR	Output a cross reference of the program to the device/file assigned to the M:LO DCB.
NCR	No cross-referenced produced.
IS	The ARRAY subscripting is calculated by inline generated code.
NIS	The ARRAY subscripting is calculated by calling a runtime library routine.
CS	All subscripts to ARRAYS are individually checked to see that they are within the defined bounds.
NCS	No checking is performed during the ARRAY subscript computation.

The options LS, NLO, NAD, NGO, FL, NBO, NDO, NDM, NBD, NSC, NCR and NIS, and CS will be assumed unless the complementary options are specified. All source will be input through the device assigned to M:SI. If FL is specified (or assumed), the compiled output in the GO file must be loaded and executed under ALOAD. The relocatable object program (ROM) may be loaded by the standard loaders and executed, like any other program. Note that specifying FL and GO is inconsistent, and should not be used together.

11.3 SOURCE CARD FORMAT

The source program is punched into cards in free format using normally columns 1 through 80 with column 80 adjacent to column 1 of the next card in sequence.

11.4 FAST LOADER

A special fast loader called ALOAD has been provided to load a binary object program in the GO file that was placed there by ALGOL using the FL option. This processor is not capable of loading any other types of program, and should not be used as such.

PROCESSOR CALL:

!ALOAD [(SL,hex)]

The SL option specified the severity level that will be tolerated in loading the program, where hex is a hexadecimal digit (between 0 and F). If the number of compile-time errors* exceeds the severity level, the program will not be executed. Otherwise, the program will begin execution immediately upon loading. If the option is not specified, the severity level will be assumed to be zero.

---

* Warning messages do not count as errors.

11.5 SAMPLE FORMS

In the following examples, the job may either be entered thru the card reader, or using the BPM subsystem. Output directed to files can be printed using FMGE in a background job, or using FERRET or EDIT in a time-sharing job.

SAMPLE FORM: (Using Fast Loader)

```
Output:    Line Printer

!JOB       Accounting Information
!ASSIGN    (if any)
!ALGOL     Options
           [ SOURCE DECK ]
!ALOAD     Option
           [ DATA (IF ANY) ]
!FIN
```

SAMPLE FORM: (Using ROM's)

```
Output:    Line Printer

!JOB       Accounting Information
!ASSIGN    M:BO,(FILE,PROG)
!ASSIGN    (if any)
!ALGOL     BO, options
           [ SOURCE DECK ]
!LOPE      (EF,(PROG),(EXEC),(UNSAT,
           (ALGOLIB))
           [ DATA (IF ANY) ]
!FIN
```

SAMPLE FORM; (Using Fast Loader)

```
Output:   Compile-time output to the printer
          Program output to a file
          Run-time error messages to the printer

!JOB      Accounting Information
!ALGOL
     [ SOURCE ]
!ASSIGN F:6,(FILE,file-name),(SAVE)
!ALOAD
!SAVE (ALL)
!FIN
```

SAMPLE FORM: (Using Fast Loader)

```
Output:   Compile-time output to file-name.
          Program output to file-name.
          Run-time error messages to file-name.

!JOB      Accounting Information
!ASSIGN   M:LO,(FILE,file-name),(SAVE)
!ALGOL
     [ SOURCE ]
!ASSIGN   M:LO,(FILE-file-name),(SAVE)
!ASSIGN   F:6,(FILE,file-name),(SAVE)
!ALOAD
!SAVE (ALL)
!FIN
```

SAMPLE FORM: (Using Fast Loader)

Source Input: User created file.

Output: Compile-time output to the printer.
Program output to file-name using a non standard DCB F:7 (in this case WRITE statements must specify the device assigned).
Run-time errors messages to file-name.

```
!JOB    Accounting Information
!ASSIGN M:SI,(FILE,file-name)1
!ALGOL
!ASSIGN F:7,(FILE,file-name)2,(SAVE)
!ASSIGN M:LO,(FILE,file-name)3,(SAVE)
!ALOAD
!SAVE (ALL)
!FIN
```

If the source input is an EDIT created file, then each line of input must not exceed 80 characters. Each line of input of an EDIT file is terminated with a line feed (X'15') character. From the compilers viewpoint, the line feed terminates the line, and is changed by the compiler to a space which is then considered as the last character of the line.

Other control characters that may appear in an EDIT created file are the horizontal tab (X'05') and the form feed (X'0C'). The horizontal tab is lexically equivalent to a single space. The form feed causes a

M:DEVICE M:LO,(PAGE)

to be executed, i.e., a skip to the top of the next page within the source listing. The form feed is lexically equivalent to a space.

APPENDIX A

FORMAT Statement

The FORMAT statement is used to specify the conversion to be performed on data being transmitted during formatted (BCD) input/output or DECODE/ENCODE operations. It is nonexecutable and may be placed anywhere in the program. In general, conversion performed during output is the reverse of that performed during input. FORMAT statements are expressed as

FORMAT ($S_1, S_2, S_3, \ldots, S_n$)

where

$n \geq 0$

S_i is either a format specification of one of the forms described below or a repeated group of such specifications in the form

$r(S_1, S_2, S_3, \ldots, S_m)$

where

$m \geq 0$.

r is a repeat count as described below.

S_j is as described above; in other words, repetitions may be nested (to ten levels).

The commas between the S_i (and S_j) are optional except where ambiguities would arise from not separating specifications. In the absence of a comma, the compiler attaches as much as possible to the left-hand specification. For example, the specifications

I23F27.13X

will be interpreted as

I23 , F27.13 , X

and not as

I2 , 3F27.1 , 3X

To obtain the latter interpretation, the commas are required.

Every FORMAT statement should be labeled so that references may be made to it by formatted input/output statements. An entire FORMAT (the parentheses and the items they enclose) may be stored in an array variable through the use of assignment statements or input statements. In this case, as described under "FORMATs Stored in Arrays", the array itself is referenced by the input/output statements.

Format specifications describe the kind or type of conversion to be performed, specific data to be generated, scaling of data values, and editing to be executed. Each integer, real, double precision, or logical datum appearing in an input/output list is processed by a single format specification, while complex data are operated on by two consecutive format specifications. Format specifications may be any of the following forms:

rFw.d	rIw	rZw	nHs	r/
rEw.d	rLw	rMw	iX	
rDw.d	rAw	r's'	Tw	
rGw.d	rRw	rs	iP	

where

the characters F, E, D, G, I, L, A, R, Z, M, H, $, quotation mark ('), X, T, P, and slash (/) define the type of conversion, data generation, scaling, editing, and FORMAT control.

r is an optional, unsigned integer[†] that indicates that the specification is to be repeated r times. When r is omitted, its value is assumed to be 1. For example,

3I6

is equivalent to

I6, I6, I6

[†]See also "Adjustable Format Specifications".

w is an optional unsigned integer[†] that defines width in characters (including digits, decimal points, algebraic signs, and blanks) of the external representation of the data being processed. If w is not present in a specification, the size of the external field depends on the characteristics of the data and the type of conversion performed. This is discussed individually under each specification.

d for F, E, D, and input G specifications, is an optional, unsigned integer[†] that specifies the number of fractional digits appearing in the magnitude portion of the external field. If d is not present, its value is assumed to be zero, and the decimal point character preceding it should not appear. That is, Ew.0 and Ew are equivalent.

For output G specifications, d is also an unsigned integer,[†] but in this context it is used to define the number of significant digits that appear in the external field; therefore, its value should not be zero.

n is an unsigned, decimal integer that defines the number of characters being processed.

s is a string of the characters acceptable to the XDS Sigma 5/7 Extended FORTRAN IV processor (see Chapter1).

i is a signed integer[†] (plus signs are optional). The function of i is described under X and P specifications.

F Format (Fixed Decimal Point)

Form:

rFw.d

Integer, real, double precision, or either part of complex data may be processed by this form of conversion. Double precision values are converted with full precision if sufficient width is specified by w, and the value of d allows for the appropriate number of digits in the fractional portion of the field.

Output. Internal values are converted to real constants, rounded to d decimal places with an overall length of w. The field is right justified with as many leading blanks as necessary. Negative values are preceded with a minus sign. Consequently, for the specification F11.4,

273.4	is converted to	273.4000
7	is converted to	7.0000
-.003	is converted to	-.0030
-442.30416	is converted to	-442.3042

When no width is specified (i.e., w is not present), the converted field contains only the number of digits necessary to express the value, plus one blank to the right of the field. Therefore, for the specification F.1,

349.5203	is converted to	349.5b
70000	is converted to	70000.0b
-22	is converted to	-22.0b

and for the specification 2F.4, the output list

.03359, -67	is converted to	.0336b-67.0000b

where b represents the character blank.

If a value requires more positions than are allowed by the magnitude of w, only w digits will appear, and the digits lost will be from the left or most significant portion of the field. This is not treated as an error condition. Thus, for the specification F4.4,

-1.22315	is converted to	2232
432034.	is converted to	0000

In order to insure that such a loss of digits does not occur, the following relation must hold true:

$w \geq d+2+n$

where n is the number of digits to the left of the decimal point.

[†]See also "Adjustable Format Specifications".

Input. Input strings may take any of the integer, real, or double precision constant forms discussed under "Numeric Input Strings". Each string will be of length w with d characters in the fractional portion of the value. If a decimal point is present in the input string, the value of d is ignored, and the number of digits in the fractional portion of the value will be explicitly defined by that decimal point. For the specification F10.3,

33	is converted to	.033
802142	is converted to	802.142
.34562	is converted to	.34562
-7.001	is converted to	-7.001

If the width w is not specified, conversion starts with the first non-blank character in the input string and ends with the first comma or blank that follows a digit or a decimal point. The comma or blank is bypassed before conversion of the next field begins. For the specification 2F.2, the string

333,.003

is converted to the values

3.33 .003

E Format (Normalized, with E Exponent)

Form:

rEw.d

Integer, real, double precision, or either part of complex data may be processed by this form of conversion. Double precision values are converted with full precision if sufficient width is specified by w and the value of d allows for the appropriate number of digits in the fractional portion of the field.

Output. Internal values are converted to real constants of the forms

.ddd...dE ee

.ddd...dE-ee

where ddd...d represents d digits, while ee or -ee is interpreted as a multiplier of the forms

$10^{\pm ee}$

Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right justified and preceded by the appropriate number of blanks. The following are examples for the specification E15.8:

90.4450	is converted to	.90445000E 02
-435739015.	is converted to	-.43573902E 09
.000375	is converted to	.37500000E-03
-1	is converted to	-.10000000E 01
.2	is converted to	.20000000E 00
0.0	is converted to	.00000000E 00

When the width w is not present in the format specification, the converted field contains only the number of characters necessary to express the value of the data, plus one blank to the right of the field. If the specification 2E.5 is used, the output list

-774.119, 1.00001977

is converted to

-.77412E␣03␣.10000E␣01␣

where ␣ represents the character blank.

The field, counted from the right, includes the exponent digits, the sign (minus or space), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or space). If a width specification is of

insufficient magnitude to allow expression of an entire value, only w digits will appear. The digits lost are from the left or most significant portion of the field. This is not treated as an error condition.

Examples:

Value	E11.4	E8.4	E6.4
-2013.55	-.2014E 04	2014E 04	14E 04
.361887	.3619E 00	3619E 00	19E 00
.000134	.1340E-03	1340E-03	40E-03

To prevent a loss of this kind, it is necessary to ensure that the relation

$w \geq d + 6$

is satisfied by the specification. Note that the above feature can be used intentionally to obtain the exponent field, which is an indication of magnitude range for any datum. For example, for the specification E3.0,

40255.034 is converted to 05,

0.0000072 is converted to -04

In the second example, the exponent is -04 rather than -05 because an Ew.d format with d = 0 causes rounding to take place at the first significant digit, which is 7 and is therefore rounded upward.

Input. The discussion "Numeric Input Strings" contains a description of the forms permissible for strings of input characters. Conversion is identical to F format conversion. In particular, input fields for conversion in E format need not have exponents specified.

Examples:

Input Value	Specification	Converted to
-113409E2	E9.6	-11.340900
-409385E-03	E.2	-4.09385
849935E-02	E10.5	.0849935
6851	E.0	6851.0

First, the decimal point is positioned according to the specification; then, the value of the exponent is applied to determine the actual position of the decimal point. In the first example, -113409E2 with a specification of E11.6 is interpreted as -.113409E02; which, when evaluated (i.e., $-.113409 \times 10^2$), becomes -11.340900.

D Format (Normalized, with D Exponent)

Form:

rDw.d

This format is similar to E format, with the exception that for output, the character D will be present instead of the character E. For example,

for E12.6, -667.334 is converted to -.667334E 03

and

for D12.6, -667.334 is converted to -.667334D 03

Input under D format is the same as for E and F formats.

G Format (General)

Form:

rGw.d

G format is the only format that may be used with any type of data, including logical. The form of conversion it performs depends on the type of the list items. For a Gw.d specification, the following table shows the equivalent format that is used when processing list items of the various types.

List Item Type	Input	Output
integer	Iw	Iw
real	Fw.d	(see below)
double precision	Fw.d	(see below)
logical	Lw	Lw

Note that, as with all other formats, complex values are processed as two separate items; the real and imaginary parts require individual specifications, and conversion occurs as shown above for real or double precision data.

For integer and logical list items, the d (in Gw.d) need not be specified; if it is present, it will be ignored. This is the only case in which d is not assumed to be zero if not specified. G format is very useful in a widthless form. When so used, the equivalent formats shown above become widthless also (see "Numeric Input Strings").

Output of Real and Double Precision Data Under G Format. The form of output conversion used with real and double precision values depends on the magnitude of the values. G format attempts to express numbers in the most natural way; that is, they are expressed in F format whenever possible, but in E format for values that are too large or too small. Specifically, d is interpreted as indicating the number of significant digits desired, and this is exactly the number of digits that will be output. If the value of the number is such that it can be expressed by placing the decimal point anywhere within or at either end of those d digits, that is what will be done, and no exponent will be appended. If, however, preceding or trailing zeros would be required to express the value correctly, F format will not be used; instead the number will be normalized and output with a following exponent.

To express this algebraically, let M represent the magnitude of the value to be output (rounded to d significant digits). Select an integer i such that

$$10^{i-1} \leq M < 10^{i} \qquad \text{(if } M = 0.0, \text{ then } i = 0\text{)}$$

Assuming a specification of Gw.d, let n = w-4 and m = d-i. Then, if $0 \leq i \leq d$, conversion takes place according to the specification

Fn.m, 4X

If i is less than 0 or greater than d, the specification used is

Ew.d

Note that when F format is used, four blanks are output following the number, in the positions where an exponent would otherwise be. In this way, numbers that are output in columns will tend to line up underneath each other in a more readable way. The following examples illustrate the effect of G format output on values of various sizes:

Value	G 10.3	G 10.1
.02639	.264E-01	.3E-01
.2639	.264	.3
2.639	2.64	3.
26.39	26.4	.3E 02
263.9	264.	.3E 03
2639.	.264E 04	.3E 04

Note that the choice of F or E format is independent of the value of the width w. If w is not large enough, digits are lost at the left as in other numeric conversions. To ensure that this will not happen, the following relation should hold true:

$w \geq d+6$

When no width is specified, the number will be followed by a single blank; values output in F form will not be followed by four blanks.

Scale factors (see "P Specifications") apply to G format only when the E form is used, not when the F form is used. This has the effect that all values output in G format are unchanged (except for rounding). It also has the effect that values output in F form with a P scale factor cannot subsequently be input using the same FORMAT; the scale

factor will take effect during input but not during output. Thus, the new value will be different from the old by a power of 10.

Note that the rounding applied to M (above) to determine whether to use E or F format is not necessarily the same rounding that is applied when the number is actually output. Consider the following case:

```
    PRINT 5, 99.76
5   FORMAT(1P, G.2)
```

In principle, F form is to be used if the value lies in the range $.1 \leq M < 100$. The value 99.76 does lie in this range, but when rounded to two digits it becomes 100., which is outside the range; so E form is used. First the unrounded value of M is normalized (.9976E 02), then the P scale factor is applied (9.976E 01), and finally this value is rounded giving 9.98E 01, which is the way it is printed. If the first rounding had been used throughout, the final value would have been 1.00E 02, which is less accurate.

I Format (Integer)

Form:

rIw

Integer, real, double precision, or either part of complex data may be processed by this form of conversion. If the width specification w is of sufficient magnitude, real and double precision values are converted in full precision. In other words, values greater than the maximum permissible size of integer data may be processed, without the truncation of the most significant digits that is normally associated with integer operations.

Output. Internal values are converted to integer constants. Real and double precision data are truncated to integer values; however, the integers may contain as many digits as are specified by w. Negative values are preceded by a minus sign, and the field will be right-justified and preceded by the appropriate number of blanks. The specification I6 implies that

273.4	is converted to	273
7	is converted to	7
-.003	is converted to	0
-44204.965	is converted to	-44204

The converted field occupies the minimum number of positions required to express the data value whenever w is unspecified. This minimum number of digits is followed by one blank. For example, for the specification 5I the output list

345.9, 70000, -2, -.999, 3030.3030

is converted to

345␣70000␣-2␣0␣3030␣

where ␣ represents the character blank.

If the magnitude of data requires more positions than is permitted by the value of the width w, only w digits appear in the external string, and the digits lost are the most significant. This is not treated as an error condition. Thus, for the specification I2,

-778801	is converted to	01

Input. External input strings may take any of the forms discussed under "Numeric Input Strings" and conversion will be identical to F format processing, with the exception that fractional portions of a value are lost through truncation. As noted above, however, the most significant digits will not be truncated. For example, the input field

45700000000000000000000000.942

processed by an I (widthless) format, into a real or double precision variable, would produce the internal value

4.57×10^{24}

L Format (Logical)

Form:

rLw

Only logical data may be processed with this form of conversion; any other data type causes an error to occur.

Output. Logical values are converted to either a T or an F character for the values "true" and "false", respectively. The T and F characters are preceded by w-1 blanks. For examples, using the specification L4,

.TRUE.	is converted to	␣␣␣T
.FALSE.	is converted to	␣␣␣F

where ␣ represents the character blank.

Specifications in which w is undefined will cause the following conversions:

.TRUE.	is converted to	T␣
.FALSE.	is converted to	F␣

Input. If a width is specified, the first T or F encountered in the next w characters determines whether the value is "true" or "false", respectively. If no T or F is found before the end of the field, the value is "false". Thus a blank field has the value "false". Characters appearing between the T or F and the end of the field are ignored, except for commas, which terminate the input string (see "Comma Field-Termination"). For example, the following input fields, processed by an L7 format, have the indicated values:

True	False
T	F
TRUE	FALSE
.TRUE.	.FALSE.
RIGHT	READ
STAFF	LEFT
24T+T42	(blank)

For widthless logical input, the field terminates at the first comma or non-leading blank. In other words, if the first non-blank character is a comma, it terminates the field; if it is not a comma, the next blank or comma will terminate the field. The first T or F encountered within the field determines the value. If neither a T nor an F appears, the field has the value "false". As above, characters appearing between the first T or F and the blank or comma are ignored.

A Format (Alphanumeric)

Form:

rAw

Output. Internal binary values are converted to character strings at the rate of eight binary digits (two hexadecimal digits) per character. The most significant digits are converted first. That is, conversion is from left to right. The number of characters produced by an item depends on the number of words of storage allocated for that type of item (see "Storage Allocation Statements", Chapter 7). Assuming standard size specifications, the following examples illustrate the form of A format conversion:

Data Type	Internal Binary/Hexadecimal	Aw	External String
integer, real, or logical	1100 1001 1101 0101 1110 0011 0101 1100 C 9 D 5 E 3 5 C	A4	INT*
		A2	IN
		A6	␣␣INT*
		A	INT*
double precision	1100 0100 1101 0110 1110 0100 1100 0010 C 4 D 6 E 4 C 2	A8	DOUBLE=2
		A6	DOUBLE
	1101 0011 1100 0101 0111 1011 1111 0010 D 3 C 5 7 B F 2	A11	␣␣␣DOUBLE=2
		A	DOUBLE=2

where ␣ represents the character blank.

As with all other format conversions, complex data are treated either as two real or as two double precision values. In each of the examples above, the first A format specifies exactly the number of characters required to express the data fully, and therefore has the same effect as the widthless form. Normally, alphanumeric information is used with integer variables. In the examples, note that when the magnitude of w does not provide for enough positions to express the data value completely, the external field is shortened from the right (least significant) portion. This is not treated as an error condition. When w has a value greater than necessary, the external character string is preceded by the appropriate number of blank characters.

When the field width is not specified, the external character string consists of only the number of positions necessary to fully express the character value of the data. The external character string is not followed by a blank.

Alphanumeric conversions are normally used to output Hollerith information that has been created in one of the following ways:

1. Previously input using an alphanumeric format (A or R)
2. Using a Hollerith constant (e.g., M = 4HABCD)
3. Using a literal constant (i.e., in a DATA statement, or passed as an argument)

It is not recommended that this form of conversion be used with random numeric values created other than as above. The reason for this is that not all the 256 possible characters that can be produced can actually be printed. The non-printable characters may, however, be useful in other contexts (e.g., on cards, or in ENCODE operations).

Input. When the width w is larger than necessary (that is, when its magnitude is greater than the number of characters associated with the data type of the corresponding list item), the list item is filled with the rightmost characters. For example, if the list item is integer, and the specification A10 is used,

ABCDEFGHIJ is converted to GHIJ

However, when the value of w is less than the number of characters associated with the data type of the list item, the most significant positions of the list item are filled with w characters, and the remainder of the positions are filled with blanks. Consequently, when the list item is double precision and the field specification is A6,

UVWXYZ is converted to UVWXYZbb

where b represents the character blank.

Naturally, if the width has a value equal to the number of characters associated with the data type of the list item, the list item is completely filled with the external field.

Widthless specifications cause the list item to be filled by the next n characters from the input string, where n is the number of characters associated with the data type of the list item. If a list contained references to a real variable, an integer variable, and a double precision variable, in that order, and a field specification of 3A were used, processing would be in the following manner:

ABCDEFGHIJKLMNOP

is converted to

ABCD EFGH IJKLMNOP

A general rule for this type of conversion is that internal values are considered to be left-justified, while external fields are considered to be right-justified.

R Format (Alphanumeric, Right-Justified)

Form:

rRw

This form of conversion is similar to A conversion, but the rule of internal justification is reversed. In other words, internal values are considered to be right-justified with leading binary zeros, whereas with A format they are left-justified with trailing Hollerith blanks.

Output. When the size of w is insufficient to allow expression of the complete internal value, R format takes characters from the rightmost (least significant) portion of the internal value. In all other respects it is identical to A format output. This difference is illustrated in the examples at the top of the following page.

Data Type	Internal Character Value	w	A Format	R Format
integer, real, or logical	INT*	4	INT*	INT*
		2	IN	T*
		6	␣␣INT*	␣␣INT*
		none	INT*	INT*
double precision	DOUBLE=2	8	DOUBLE=2	DOUBLE=2
		6	DOUBLE	UBLE=2
		10	␣␣DOUBLE=2	␣␣DOUBLE=2
		none	DOUBLE=2	DOUBLE=2

where ␣ represents the blank character.

Input. As on output, R format differs from A format only when the specified width (w) is less than the number of characters associated with the type of the input list item. In this case, R format fills the least significant (rightmost) portion of the list item with w characters from the input string, preceded by enough binary zeros to fill the remaining portion. In other words, R format right justifies the characters and inserts leading binary zeros, while A format left justifies the characters and inserts trailing Hollerith blanks. For example,

List Item Data Type	External String	w	Internal after A Conversion	Internal after R Conversion
integer, real, or logical	XYINT*	4	XYIN	XYIN
		6	INT*	INT*
		2	XY␣␣	zzXY
		none	XYIN	XYIN
double precision	85DOUBLE=2	8	85DOUBLE	85DOUBLE
		6	85DOUB␣␣	zz85DOUB
		10	DOUBLE=2	DOUBLE=2
		none	85DOUBLE	85DOUBLE

where

␣ represents the Hollerith character blank and

z represents eight binary zeros.

Note that the Hollerith character zero is not represented internally as eight binary zeros. Consequently, if the external field

00ABAB

were processed by the format specifications A4,R2 into two integer variables, the resulting values would be the Hollerith constants 4H00AB and 2RAB, which are not equivalent. For input as true right-justified integers, R format should be used.†

Z Format (Hexadecimal)

Form:

rZw

Z conversion is similar to R conversion, except that the internal data is processed 4 bits at a time instead of 8, and the external field consists of hexadecimal digits, which are:

0 1 2 3 4 5 6 7 8 9 A B C D E F

†See, also, "Character Manipulation in FORTRAN", Appendix E.

Output. Internal binary values are converted to hexadecimal digit strings at the rate of 4 bits per digit. The number of characters produced by an item depends on the number of words of storage allocated for that type of item (see "Storage Allocation Statements", Chapter 1); For example, an integer produces 8 digits, a double precision number, 16.

If w is not specified large enough, the leftmost digits are lost, as in other numeric formats. If w is larger than the number of positions necessary to express the data, the digits are right-justified in the field, with preceding blanks.

When field width permits, all of the digits in an item are output, including leading zeros.

When w is not specified, the full number of digits necessary to express the value is output, followed by a blank. The blank is to facilitate subsequent rereading of the value (see below).

Examples:

Data Type	Internal Binary/Hexadecimal								Zw	External String
integer, real, or logical	0000	0000	0000	1000	1110	0011	0101	1100	Z8	0008E35C
	0	0	0	8	E	3	5	C	Z6	08E35C
									Z10	␣␣0008E35C
									Z	0008E35C
double precision	0100	0001	0011	0010	0100	0011	1111	0110	Z16	413243F6A8885A30
	4	1	3	2	4	3	F	6	Z11	3F6A8885A30
	1010	1000	1000	1000	0101	1010	0011	0000		
	A	8	8	8	5	A	3	0	Z18	␣␣413243F6A8885A30
									Z	413243F6A8885A30

where ␣ represents a blank character.

Input. When the width w is larger than necessary (i.e., when its magnitude is greater than the number of digits associated with the data type of the corresponding list item), the list item is filled with the rightmost characters in the field.

When w is too small, the digits are right-justified in the list item, as with R format. As usual, when the width exactly corresponds to the number of digits associated with the list item, the item is completely filled with the external field.

There is, however, a significant difference between Z and R format on input. Z format is a numeric format, not alphanumeric. Therefore, commas may be used to terminate a hexadecimal input string. Furthermore, the length of a widthless Z input string is not dependent on the size associated with the list item; a widthless hexadecimal input string terminates at the first comma or non-leading blank, like all other numeric formats. Excess digits will be lost at the left. (Note that, when w is specified, blanks are treated as zeros.)

The following are examples of Z format input (assuming an integer list item):

External Input Field	Format	Internal Hexadecimal Value
3A70049B	Z5	0003A700
␣␣␣3A7␣␣	Z8	0003A700
␣D68,47019	Z8	00000D68
DCBA987654321	Z12	98765432
52CA91,	Z	0052CA91
123456789ABC,	Z	56789ABC
␣␣49␣␣3	Z	00000049

where ␣ represents a blank.

M Format (Machine Dependent)

M format is intended primarily for output. It provides a machine-independent method of dumping information in the format most appropriate to the machine on which the program is running. Thus, on an octal machine it would

be interpreted as O format, and on a character machine, as A format. On the Sigma 5/7, it is interpreted exactly the same as Z (hexadecimal) format. Thus, it could also be used for input, though this is not recommended.

H Format (Hollerith)

Form:

nHs

where

$n \leq 255$

Output. The n characters in the string s are transmitted to the external record. For instance,

Specification	External String
1HE	E
8Hbbbb VALUE:	bbVALUE:
5H$3.95	$3.95
9HX(2,5)b=b	X(2,5)b=b

where b represents the character blank.

Care should be taken that the character string s contains exactly n characters, so that the desired external field will be created, and so that characters from other format specifications are not used as part of the string.

Input. The n characters in the string s are replaced by the next n characters from the input record. This replacement occurs as shown in the following examples:

Specification	Input String	Resultant Specification
3H123	ABC	3HABC
10HNOWbISbTHE	bTIMEbFORb	10HbTIMEbFORb
5HTRUEb	FALSE	5HFALSE
6Hbbbbbb	RANDOM	6HRANDOM

where b represents the character blank. This feature can be used to change the titles, dates, column headings, etc., that are to appear on an output record generated by the H specification.

If n is not present, its value is assumed to be 1.

' and $ Formats (Hollerith)

These are alternate formats for Hollerith transmission similar to that done by H format. They have the advantage of not requiring the characters in the string to be counted.

Form:

's'

s

The string s may contain not more than 255 characters. Any Hollerith characters may appear (see Chapter 1); however, note the restrictions below concerning the ' and $ characters themselves. A repeat count, r, may optionally precede either of these specifications.

Output. The string s is transmitted to the external device in a manner similar to that for H format. Thus,

'ABLE ',$BODIED$

is output as the string

ABLE BODIED

Within a ' or $ string, these characters themselves are represented by a double ' or $, respectively.[†] For example, within a ' string, two consecutive ' characters[†] are interpreted as one ' character, as in

'''TWAS BRILLIG AND THE SLITHY TOVES...'

[†]See the footnote on the following page.

Within a ' string, a $ character has no special significance, and vice versa. For example, the specifications

```
$I'LL $ , 'TAKE $'
```

produces the external string

```
I'LL TAKE $
```

Input. The characters appearing between the quotes or dollar signs are replaced by the same number of characters, taken sequentially from the input string. Therefore, if the specification

```
'VECTOR'
```

is used to process the input field

```
MATRIX
```

the specification itself is changed to

```
'MATRIX'
```

Normally, dollar signs should not appear in the input string for a $ format, nor should quote (') characters appear in the input string for a ' format; if they do appear, they will be changed to blanks.

With one exception, blanks in FORMAT statements are significant only in H, ', and $ specifications.[†]

X Specification (Skip; Space or Backspace)

The form of the X specification is

```
iX
```

This specification causes no conversions to occur. Instead, it causes i positions of the external field to be "skipped". If i is positive, it has an effect similar to that of a space bar on a typewriter; if it is negative, it has an effect similar to that of the backspace control on a typewriter. In particular, an attempt to backspace beyond the beginning of a record is equivalent to backspacing to the beginning of the record.

Output. For positive values of i, the next i positions in the output record will be blanks (normally; however, see below). In other words, a field of i blanks will be created. For example, the specifications

```
'WXYZ' , 4X , 'IJKL'
```

generate the following external string:

```
WXYZbbbbIJKL
```

where b represents the character blank.

A negative value of i causes processing to "back up" in the record. The next field will then begin |i| characters to the left, assuming that this is not beyond the beginning of the record. For example, the specifications

```
'FORTRAN' , -3X , 'KNOX'
```

are equivalent to the specification

```
$FORTKNOX$
```

[†]Note that two ' or $ characters are not considered consecutive if they are separated by a blank. Therefore, the two FORMAT statements

```
1 FORMAT($ABC$    $DEF$)
2 FORMAT($ABC$$DEF$)
```

are not equivalent. They produce the following external strings, respectively:

```
ABCDEF
ABC$DEF
```

This is the only exception to the rule that, within a FORTRAN program, blanks are significant only within H, ', and $ strings; in the first FORMAT statement above, the blanks between the dollar signs are not within the string, but they cannot be removed without changing the meaning of the statement.

Note that when either backing up or moving forward by means of an X specification, characters that may have been previously produced in the positions being skipped are not destroyed. Thus, in the example given above under X output, it is not necessarily true that the specification 4X will produce four blanks. It will, however, if no other characters have been generated in those positions, since all output records are initially set to blanks.

The ability to specify a negative count in an X specification makes it possible to backspace over the blank that is produced at the end of external fields by widthless numeric formats (i.e., D, E, F, G, and I). For example, for K = 13 and Q(13) = 350.8, the statements

```
  PRINT 5,  K  ,  Q(K)
5 FORMAT( 'Q(' , I , -X , ') = ' , F.2)
```

generate the string

```
Q(13) = 350.80
```

As illustrated in the above example, if i is not specified it is assumed to be 1. Thus, the following specifications are equivalent:

```
XXXX
4X
```

Input. The next i characters from the input string are ignored whenever i is positive (that is, they are skipped). For example, with the specifications

```
F5.3, 6X, I3
```

and the input string

```
76.41IGNORE697
```

the characters

```
IGNORE
```

will not be processed.

Negative values of i cause |i| characters from the input string to be processed again. Consequently, the specifications

```
I3, -1X, E4.1
```

and the string

```
123456
```

are equivalent to

```
I3, E4.1
```

and the string

```
1233456
```

T Specification (Tab)

The form of the T specification is

```
Tw
```

This specification causes processing (either input or output) to begin at character position w in the record, regardless of the position in the record that was being processed before the T specification. It functions exactly like an X specification; no transfer of data occurs. For example, the following FORMATs are equivalent:

```
1 FORMAT( 5X , A8 , -2X , I7 )
2 FORMAT( T6 , A8 , T12 , I7 )
```

It can be seen from the above example that it is permissible to tab either forward or backward. Furthermore, a T specification provides a capability that an X specification does not, namely that of tabbing to a given print position

when widthless formats are being used and the character position is thus unknown. For example, to print (or read) three columns of numbers beginning in positions 1, 21, and 41, the following FORMAT statement could be used:

3 FORMAT(G.7,T21,G.7,T41,G.7)

Note that backward tabbing can cause previously output information to be overwritten, or previously read input to be processed again.

As with X specifications, it is not possible to tab to a position previous to the beginning of the record.

If no w is given, it is assumed to be 1. That is, T is the same as T1.

P Specification (Scale Factor or Power of 10)

The form of the P specification is

iP

A P specification causes the value of the scale factor to be set to i, where the scale factor is treated as a multiplier of the forms

10^{i} for output

and

10^{-i} for input

At the beginning of each formatted input/output operation, before any processing occurs, the scale factor is set to zero. Any number of P specifications may be present in a FORMAT statement, thereby causing the value of the scale factor to be changed several times during a formatted input/output operation. If a FORMAT is re-scanned within a single input/output operation due to the number of items in a list (see "FORMAT and List Interfacing"), the value of the scale factor is not reset to zero.

Scale factors are effective only with F, E, and D conversions, floating-point input G conversions, and E-type output G conversions.

Output. The value of the list item is scaled by the multiplier 10^{i}. This scaling causes the decimal point to be shifted right i places. On D- and E-type conversions, the exponent field (±ee) is correspondingly reduced by 1. Thus, for D- and E-type output, the external number is equal to the internal value (except for rounding), while for F format output it is not (unless i is 0). Scale factors do not affect numbers whose value is zero. The following examples illustrate output scaling:

Format	External field when internal value is:			
	2.71828	-2.71828	0.00000	0.09999
2PF10.3	271.828	-271.828	.000	9.999
1PF10.3	27.183	-27.183	.000	1.000
0PF10.3	2.718	-2.718	.000	.100
-1PF10.3	.272	-.272	.000	.010
-2PF10.3	.027	-.027	.000	.001
-3PF10.3	.003	-.003	.000	.000
-4PF10.3	.000	-.000	.000	.000
2PE14.3	27.183E-01	-27.183E-01	.000E 00	99.990E-03
1PE14.3	2.718E 00	-2.718E 00	.000E 00	9.999E-02
0PE14.3	.272E 01	-.272E 01	.000E 00	.100E 00
-1PE14.3	.027E 02	-.027E 02	.000E 00	.010E 01
-2PE14.3	.003E 03	-.003E 03	.000E 00	.001E 02
-3PE14.3	.000E 04	-.000E 04	.000E 00	.000E 03
-4PE14.3	.000E 05	-.000E 05	.000E 00	.000E 03

The examples for E conversion above are similar to those that would result from D conversion and E-type G conversion. When G conversion uses the F form, however, scale factors do not apply. Thus, a number output in G format always represents the internal value.

Note that when a scale factor is in effect, output rounding takes place after the scaling has been performed. In the case of E format, this may cause additional scaling to be required, as shown above in the output of 0.09999. Note the discontinuity in the way the exponent changes.

Input. During F, E, D, and G input conversions, if the input string contains an exponent field, the scale factor has no effect. However, when the input string does not contain an exponent field, the value of the external field is scaled by 10^{-i}; that is, the decimal point is moved left i places. The following examples indicate the effect of scaling during an input operation:

External Field	Scale Factor	Effective Value
-71.436	0P	-71.436
	3P	-.071436
	-1P	-714.36
-71.436E 00	3P	-71.436
	-1P	-71.436

It can be seen that, on both input and output, if the external number has an exponent specified, it is equal to the internal value; if it does not, then

external value = internal value x 10^{i}

Once a scale factor has been established during an input/output operation, it remains in effect throughout the operation, unless redefined by an additional P specification. To reset the scale factor to zero, it is necessary to write a 0P specification. For the list

A, K, X, B

and the FORMAT

FORMAT(2(F.3, 2P), E12.4, -2P)

A, K, and B are all converted using the F.3 format specification, but all three have different scale factors in effect, as illustrated below:

List Item	Effective Format Specification
A	F.3
K	2PF.3
X	2PE12.4
B	-2PF.3

When i is not specified, its value is assumed to be zero. Therefore,

P is equivalent to 0P

/ Specification (Record Separator)

The form of the / specification is

r/ or /

Each slash(/) specified causes another record to be processed. In the case of contiguous slash specifications (i.e., ////.../ or r/), since no conversion occurs between each of the slash specifications, records are ignored during input, and blank records are generated during output operations. The same condition can occur when a slash specification and either of the parenthesis characters surrounding the field specifications are contiguous; a slash preceding the final right parenthesis in a FORMAT statement is not ignored.

Output. Whenever a slash specification is encountered, the current record being processed is output, and another record is begun. If no conversion has been performed when the slash is encountered, a blank record is created. The statements

```
      WRITE (5, 10) X, Y
   10 FORMAT (F5.3//I13)
```

are processed in the following manner:

1. A record is begun, and X is converted with the specification F5.3.
2. The first slash is encountered, the record containing the external representation of X is terminated, and another record is begun.
3. The second slash is encountered, the second record is terminated, and a third record is started. Note that since no conversion occurred between the terminations of the first and second records, the second record was blank.
4. The value of the variable Y is converted with the I13 specification, the closing right parenthesis character is encountered, and the third record is terminated.

If a third item Z were added to the output list, as in

```
      WRITE (5, 10) X, Y, Z
```

the following additional steps would occur:

5. A fourth record is begun, and Z is converted using the specification F5.3.
6. The first slash is re-encountered, the fourth record is terminated, and a fifth record is begun.
7. Again, the second slash is processed; the fifth record, which is blank, is terminated; and the sixth record is started.
8. Since there are no more list items, the specification I13 is not processed, a termination occurs, and the final or sixth record, which is also blank, is output.

Note that the processing of Z in steps 5 through 8 is equivalent to processing with the statement

```
   10 FORMAT (F5.3, //)
```

since the specification I13 was not utilized.

The original FORMAT statement could also have been written as

```
   10 FORMAT (F5.3, 2/I13)
```

or

```
   10 FORMAT (F5.3, 2/, I13)
```

both of which would cause identical effects.

The two statements

```
      WRITE (M, 4) X
    4 FORMAT (3/E.4/)
```

cause the generation of three blank records, followed by a record containing the value of X (converted by the specification E.4), followed by another blank record.

Input. The effect of slash specifications during input operations is similar to the effect for output, except that for input, records are ignored in the cases where blank records are created during output. For example, the statements

```
      READ (M, 4) X
    4 FORMAT (3/E.4/)
```

cause three records to be bypassed, a value from the fourth record to be converted (with the specification E.4) and assigned to X, and a fifth record to be bypassed. This means that, as with the last example for output, records created with a FORMAT statement containing slash specifications can be input by use of the identical FORMAT statement, which is not true in FORTRAN systems that ignore a final slash.

Parenthesized Format Specifications

Within a FORMAT statement any number of specifications may be repeated by enclosing them in parentheses, preceded by an optional repeat count, in the form shown on the following page.

$r(S_1, S_2, S_3, \ldots, S_m)$

where r and the S_i are defined previously, and $m \geq 0$. For example, the statement

```
3 FORMAT (3(A4, F.2, 3X), 3I)
```

is equivalent to

```
3 FORMAT (A4, F.2, 3X, A4, F.2, 3X, A4, F.2, 3X, 3I)
```

There is no limit to the number of repetitions of this form that can be present in a FORMAT statement.

During input/output processing each repetitive specification is exhausted in turn, as is each singular specification.

The following are additional examples of repetitive specifications:

```
  34 FORMAT (4X, 2(A8, X, 7G.3), I4, 3(D, L5))
1125 FORMAT (/, R4, F.7, 5(E14.8, 2/), E14.8)
   8 FORMAT (2(I8, 2(3X, F12.9), F12.9), A16)
```

In the last example above, repetitions have been nested. Nesting of this type is permissible to a depth of ten levels.

The presence of parenthesized groups within a FORMAT statement affects the manner in which the FORMAT is rescanned if more list items are specified than are processed the first time through the FORMAT statement. In particular, when one or more such groups have appeared, the rescan begins with the group whose right parenthesis was the last one encountered prior to the final right parenthesis of the FORMAT statement. A more complete discussion of this process is contained in "FORMAT and List Interfacing".

Adjustable Format Specifications

This is one of the most powerful features of XDS Sigma 5/7 Extended FORTRAN IV. It often eliminates the need to write a great number of FORMAT statements in order to handle slightly different situations. Furthermore, it facilitates the input of records whose form is highly variable, and which could not be processed without this feature.

Any of the quantities r, w, d, or i (see "FORMAT Statement") may be replaced by the letter N in a format specification. When an N is encountered, its value is obtained from the next input or output list item. The letter N is merely a form of specification and does not conflict with any variable, subprogram, etc., whose identifier may be N. Also, there is no limit to the number of N characters that may be used in a FORMAT statement or to the number of quantities replaced by N in a format specification. For example,

```
32 FORMAT (NX, FN.4, N(3X, E.5), NP, NGN.N)
```

is a valid statement, and seven values will be taken from the list.

The following set of rules defines the manner in which the value of N must be specified in a list and the way in which the values are utilized:

1. Integer, real, double precision, or either part of complex data may be supplied as values for N. Non-integer data will be truncated to integer value.
2. w and d (width and decimal point) specifications may be replaced only by N, whereas r (repeat count) and i (skip or scale factor count) specifications may be replaced by N or -N.
3. The resultant value (negated if preceded by a minus sign) may be negative only when N is used to replace i.
4. When N appears one or more times in a single specification, its values must appear sequentially in the list and prior to the items (if any) that are to be processed by the specification. An example is the list

   ```
   3, 4, 1, A, B, C, 12, -2, D
   ```

   and the statement

   ```
   3 FORMAT (NEN.N, NX, NP, G14.8)
   ```

   which are equivalent to the list

   ```
   A, B, C, D
   ```

   and the statement

   ```
   3 FORMAT (3E4.1, 12X, -2P, G14.8)
   ```

5. Whenever N is used with a specification that is enclosed in repetition-type parentheses (see "Parenthesized Format Specifications"), one value must be supplied for each repetition of the specifications enclosed. Consequently, the difference between the following two examples should be noted:

7, A, B, C and 3FN.2	7, A, 7, B, 7, C and 3(FN.2)
are equivalent to	are equivalent to
A, B, C and 3F7.2	A, B, C and 3F7.2

6. In the above example, it was noted that in the specification 3FN.2, one value of N is required, regardless of the value of the repeat count; whereas, in 3(FN.2), the number of values required for N is equal to the repeat count. The same rule can be extended to include repeat counts whose values are zero:

   a. When the repeat count (r) of a single specification is replaced by N and its value is zero, any Ns appearing in <u>that</u> specification must be supplied. For example, the following combination of list and FORMAT,

      0, 4, Y and NG20.N, F8.4

      are equivalent to

      Y and F8.4

   b. However, when the repeat count of a parenthesized group is replaced by N and its value is zero, all the specifications appearing within the parentheses are bypassed, including any Ns that may appear. Thus,

      0, Y and N(G20.N), F8.4

      are equivalent to

      Y and F8.4

   In both of the above examples, no value was supplied for the G specification; however, enclosing the specification within parentheses can be used to determine whether or not the value of N will be supplied.

   The ability to specify zero repeat counts in this way gives the programmer the facility of selecting or skipping certain specifications within a FORMAT statement. For example,

```
T = 0
F = 1
IF (BOOLE) T = 1 ; F = 0
PRINT 17, BOOLE , T, F
17 FORMAT(L1 , N(3HRUE) , N$ALSE$)
```

   outputs the strings TRUE or FALSE depending on the value of BOOLE. Note that although an N cannot replace the n in an H specification, the form shown in statement 17 above can be used.

7. The value of N may be supplied by an expression in either an input or an output list, but an expression used for this purpose in an <u>input</u> list is not considered to be a true input list item.

As an example of the flexibility provided by adjustable format specifications, consider the statements

```
READ(101,205) K, K, (A(J), J=1,K), CODE
205 FORMAT( I , NE , A4 )
```

The value input for K defines not only the number of values to be input into the array A, but also the number of conversions to be performed by the E specification. At the same time, the alphanumeric value of CODE can be contiguous to the last field input into A, regardless of the number of such fields. Thus, all the following input records can be correctly processed by the above statements:

```
1, 67.49,HOPA
5 -14.3 37 .09711623 0 3E12 JASU
,NONE
```

This example illustrates not only adjustable format specifications, but also widthless formats and comma field termination (see below).

Numeric Input Strings

The permissible kinds of input strings that may be processed by numeric conversions are exactly the same for F, E, D, G, and I conversion. Any field that can be read using one of these formats can be read using any of the others. In other words, numbers for input with E format need not have exponents, numbers for input with I format need not be integers, etc.

A numeric input string consists of a string of digits with or without a leading sign, a decimal point, and/or a trailing exponent. An exponent is normally specified as

E±e

where the plus sign is optional and e is a one- or two-digit number. The form ±e is also accepted (without the E), in which case the plus sign is not optional. Thus, a variety of forms may be used to express data for numeric input:

±n	±n.m	±n.	±.m
±nE±e	±n.mE±e	±n.E±e	±.mE±e
±n±e	±n.m±e	±n.±e	±.m±e

where the plus signs are optional except in an exponent field without an E (as described above).

When input fields contain no decimal point (as in the first column above) the decimal point is positioned according to the d in the format specification (as in Ew.d). If none is specified it is assumed to be zero. The decimal point is placed d positions to the left of the beginning of the exponent, or if no exponent is present, d positions to the left of the end of the field. Note that the exponent may begin with either a D, E, +, or -.

A D may be substituted for the E in an exponent field, with no change in meaning or value. It is not necessary to indicate that data is double precision, nor is it necessary to use a D format. Regardless of the format used or the form of exponent (if any), a numeric string will be converted with full double precision if the input list item to which it is to be assigned is double precision.

Any numeric type of list item may be used with any numeric type of format specification. If the list item is integer, the input value will be processed in floating-point, if necessary, and then converted to integer. When the I format specification is used (with any type of list item), the fractional portion of the value is lost.

A comma may be used to terminate any numeric field, as described below. Leading blanks are always ignored. The interpretation of embedded and trailing blanks depends on whether or not the format specification used is widthless (no width specified).

Widthless Numeric Input

The principle behind widthless input is that the field ends when the number is finished. A comma always indicates that the number is finished. A blank also indicates that the number is finished, if it is meaningful to finish the number at that point. Thus:

1. Leading blanks do not cause termination; they are ignored.

2. Any number of blanks may appear in the following places:
   a. Between the leading plus or minus sign and the first digit.
   b. Between the E and the plus or minus sign or first digit of the exponent.
   c. Between the plus or minus sign in the exponent and the first digit of the exponent.

3. A blank that follows a digit or decimal point terminates the field.

4. When a widthless (or any other) field runs off the end of the input record, the extra characters will be interpreted as blanks. Normally, a widthless format does not terminate until at least one non-blank character has been found. Special provision is made, however, to terminate widthless fields at the end of the record. Thus, any number of numeric values may be read from a blank record, and they will all be zero.

For clarity, numbers should generally be written without any embedded blanks. The first blank will then terminate the field. Although the terminating blank or comma does not afffect the value of the number, it is considered part of the field it has terminated. Therefore, the next field begins with the character following the blank or comma.

The following is a typical widthless numeric input line consisting of eight values:

73 2E-4 .0007 -35.4 0 0 -16 27.08614E 12

The following is not a typical widthless numeric input line:

- 3E + 2 + 3.7 - 4 17E 2 5- 03

but would be interpreted as five values, namely,

-300. 3.7 -4. 1700. .005

Numeric Input with Width Specified

When a width is specified, the field terminates only when the width is exhausted or a comma is found. The following rules apply to blanks in numeric fields with a width specified:

1. Leading blanks are ignored, except that they are counted as part of the field width.
2. Once any nonblank character has been found, all blanks beyond that point are treated as zeros.
3. Any string of digits that is omitted has an assumed value of zero.

For a format specification such as F10.0, with no P scale factor, all the input strings in each of the columns below produce the value shown in the top line of the column. The first three lines in each column are typical numeric fields; the others are permissible, but less readable.

-.004	7.5E12	0
-4E-3	.75D+13	0.0
-.004	750E10	
- 4 - 4	75 E1	0 + 0
- . 4D	75 + 01	0E
-4 -8	. 75E16	+ -

Note, in the fourth example of the middle column above, that the exponent is interpreted as 10 rather than as 1, because the trailing blank is equivalent to a zero. Care should always be taken to assure that exponents are right-justified in their fields. Failure to do this is a common pitfall that can also be avoided by using comma termination and/or widthless formats.

Comma Field-Termination

Input strings being processed under control of F, E, D, G, I, or L specifications may be terminated at any point by the presence of a comma in the string.† In other words, whenever a comma appears in such an input string, the field currently being processed is considered ended, and no additional characters are converted. This termination occurs regardless of the value of w in the field specification. The comma is not processed, and the next field begins with the character following the comma. For example, the specification 2F13.3 and the string

3450,88412,

are equivalent to F4.3,F5.3 and

345088412

The string containing the commas would also be correctly processed by the specifications 2F.3 or 2F8.3.

Two contiguous comma characters indicate an empty field, which has the value zero. Therefore, for the specification 5I6, the string

303,,-1,,000450

is converted to the values

303 0 -1 0 450

†For consistency with symbolic input (via the INPUT statement), the characters semicolon, asterisk, and right parenthesis are also accepted as field terminators. Use of the comma is recommended, however.

and the string

```
0,,,,,
```

is converted to the values

```
0  0  0  0  0
```

The comma must, of course, fall within the field it is meant to terminate. For example, if the format specification F4.0 were used to process the input string

```
1234,
```

the value would already be terminated because of field width, and the comma would terminate the following field, giving it a value of zero.

FORMAT and List Interfacing

Formatted input/output operations are controlled by the FORMAT requested by each READ or WRITE statement. Each time a formatted READ or WRITE statement is executed, control is passed to the FORMAT processor. The FORMAT processor operates in the following manner:

1. When control is initially received, a new input record is read, or construction of a new output record is begun.

2. Subsequent records are started only after a slash specification has been processed (and the preceding record has been terminated) or the final right parenthesis of the FORMAT has been sensed. Attempting to read (or write) more characters on a record than are (or can be) physically present does not cause a new record to be begun; on output the extra characters are lost, on input they are treated as blanks.

3. During an input operation, processing of an input record is terminated whenever a slash specification or the final right parenthesis of the FORMAT is sensed, or when the FORMAT processor requests an item from the list and no list items remain to be processed. Construction of an output record terminates, and the record is written on occurrence of the same conditions.

4. Every time a conversion specification (i.e., F, E, D, G, I, L, A, R, Z, M, or N specification) is to be processed, the FORMAT processor requests a list item. If one or more items remain in the list, the processor performs the appropriate conversion and proceeds with the next field specification. (If conversion is not possible because of a conflict between a specification and a data type, an error occurs.) If the next specification is one that does not require a list item (i.e., H, $, ', X, T, P, or /), it is processed whether or not another list item exists. Thus, for example, the statements

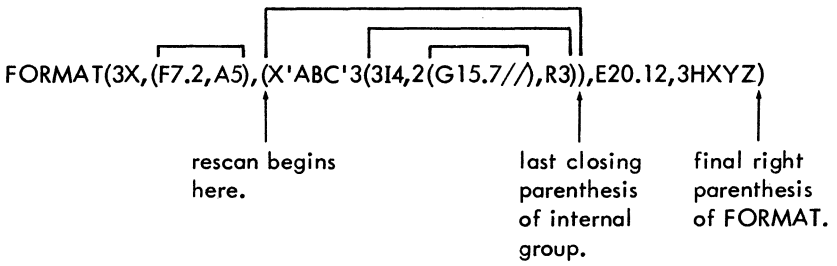
   ```
   WRITE(6, 12)
12 FORMAT(///4HABCD)
   ```

   would produce three blank records and one record containing ABCD before reaching the final right parenthesis. When there are no more items remaining in the list and the final right parenthesis has been reached or a conversion specification has been found, the current record is terminated, and control is passed to the statement following the READ or WRITE statement that initiated the input/output operation.

5. When the final right parenthesis of a FORMAT statement is encountered by the FORMAT processor, a test is made to determine if all list items (including those to be used as values of N in adjustable specifications) have been processed. If the list has been exhausted, the current record is terminated, and control is passed to the statement following the READ or WRITE statement that initiated the input/output operation. However, if another list item is present, an additional record is begun, and the FORMAT statement is rescanned. The rescan takes place as follows:

   a. If there are no parenthesized groups of specifications within the FORMAT statement, the entire FORMAT is rescanned.

   b. If one or more parenthesized groups do appear, however, the rescan is started with the group whose right parenthesis was the last one encountered prior to the final right parenthesis of the FORMAT statement. In the following example, the rescan begins at the point indicated.



c. If the group at which the rescan begins has a repeat count (r) in front of it, the previous value of the repeat count is used again for each rescan. In particular, if the repeat count was specified with an N, a new value of N is not supplied when the rescan takes place; the old value is used. Thus for example, the statements

```
      PRINT 5, CODE, 5, (A(J), J=1,50)
    5 FORMAT( A4 / N(G20.8) )
```

are equivalent to the statements

```
      PRINT 5, CODE, (A(J), J=1,50)
    5 FORMAT( A4 / 5(G20.8) )
```

6. Each list item to be converted is processed by one specification or one iteration of a repeated specification, with the exception of complex data, which are processed by two such specifications.

7. Each READ or WRITE statement containing a non-empty list must refer to a FORMAT statement that contains at least one adjustable or conversion (see step 4 above) specification. If this condition is not met, the FORMAT statement will be processed, but an error will occur.

8. The same rules apply to DECODE and ENCODE operations as to READ and WRITE. The interpretations of multiple records in these cases is described under "Memory-to-Memory Data Conversion".

APPENDIX B

Backus Normal Form for Input and Output

1. READ statement

```
<READ statement> ::= READ(<input list>)|
        READ(<DCB number>,<input list>)|
        READ(<format designator>,<input list>)|
        READ(<DCB number>,<format designator>,<input list>)
<DCB number> ::= <integer>|<integer variable>
<format designator> ::= <format label>| BINARY
<format label> ::= <identifier>
<input list> ::= <input item>|<input list>,<input item>
<input item> ::= <variable>|<array identifier>
```

2. WRITE statement

```
<WRITE statement> ::= WRITE(<format label>)|
        WRITE(<output list>)|
        WRITE(<DCB number>,<output list>)
        WRITE(<format designator>,<output list>)
        WRITE(<DCB number>,<format designator>,<output list>)

<output list> ::= <output item>|<output list>,<output item>
<output item> ::= <variable>|<number>|<array identifier>|
<expression>
```

3. OPENREAD statement

```
<OPENREAD statement> ::= OPENREAD|
        OPENREAD(<format designator>)|
        OPENREAD(<DCB number>)|
        OPENREAD(<DCB number>,<format designator>)
```

4. OPENWRITE statement

```
<OPENWRITE statement> ::= OPENWRITE|
        OPENWRITE(<format designator>)|
        OPENWRITE(<DCB number>)
        OPENWRITE(<DCB number>,<format designator>)
```

5. INPUT statement

   <INPUT statement> ::= INPUT(<input list>)

6. OUTPUT statement

   <OUTPUT statement> ::= OUTPUT(<output list>)

7. CLOSEREAD statement

   <CLOSEREAD statement> ::= CLOSEREAD

8. CLOSEWRITE statement

   <CLOSEWRITE statement> ::= CLOSEWRITE

9. FORMAT declaration

   <FORMAT declaration> ::= FORMAT<format label> <format specification><format specification> ::= <XDS-FORTRAN format specification>

10. EOF statement

    <EOF statement> ::= EOF|EOF(<designation expression>)

11. ERR statement

    <ERR statement> ::= ERR|ERR(<designational expression>)

12. REWIND statement

    <REWIND statement> ::= REWIND(<DCB number>)

13. BACKSPACE statement

    <BACKSPACE statement> ::= BACKSPACE(<DCB number>)

14. ENDFILE statement

    <ENDFILE statement> ::= ENDFILE(<DCB number>)

APPENDIX C

Predefined Identifiers

The following is a list of identifiers that may not be used in any context other than that set down in this manual. They are predefined before the compilation process begins.

AND	ELSE	INPUT	READ
ARRAY	END	INTEGER	REAL
BACKSPACE	ENDFILE	LABEL	REWIND
BEGIN	EOF	LEQ	STEP
BINARY	EQIV	LOCAL	STRING
BOOLEAN	ERR	MOD	SWITCH
CAT	EXTERNAL	NEQ	THEN
CLOSEREAD	FALSE	NOT	TRUE
CLOSEWRITE	FOR	OPENREAD	UNTIL
COMMENT	FORMAT	OPENWRITE	VALUE
COMPLEX	GEQ	OR	WHILE
DEFINE	GO	OUTPUT	WRITE
DO	IF	OWN	XOR
DUMP	IMPL	PROCEDURE	

The following is a list of identifiers that are predefined before the compilation process begins, but may be declared in any block. Outside the scope of the declaration, the identifier reverts to the meaning set down in this manual.

ABS	COMPLX	LENGTH	SIN
ARCTAN	ENTIER	LN	SQRT
CONJ	EXP	RE	
COS	IM	SIGN	

APPENDIX D

Error Messages

1. COMPILE-TIME ERROR MESSAGES

When a compile-time error occurs, the compiler attempts to recover from the error and continue. However, in doing so, parts or entire statements may be bypassed. This can lead to additional errors that are not actually present, and they will disappear once the preceding errors are corrected. For example, during the recovery process a BEGIN may be bypassed; consequently its matching END will match another BEGIN. This situation could result in prematurely closing out a procedure or block, thus making certain variables no longer defined.

If an undefined identifier is used arithmetically, an appropriate error message is given. Then in order to avoid redundant error messages, the compiler automatically defines the identifier to be a REAL quantity. In all other contexts, an undeclared identifier is defined as a label.

0: INTERNAL ERROR
- Should never occur. If it does, bring your listing to one of the Computer Center staff members.

1: IMPROPER CONSTANT
- Integer too large, i.e., it is greater than $2^{32}-1$.
- Number misformed, i.e., it is not syntactically correct.
- Real number too large, i.e., it is greater than 7.237×10^{75}.
- Real number too small, i.e., it is less than 5.398×10^{-79}.

2: IMPROPER DECLARATION
- Declaration does not occur in block head.
- Syntax of declaration is incorrect.
- VALUE declaration appears outside a procedure heading.
- OWN or LOCAL declaration occurs in a procedure specification part.
- Identifier in VALUE part not a formal parameter.
- Arithmetic specified for a procedure in a LOCAL declaration conflicts with that specified in the procedure's declaration.

3: IMPROPER OWN DECLARATION
- Declaration does not occur in the block head.
- OWN not followed by the word INTEGER, REAL, BOOLEAN, ARRAY, STRING or COMPLEX.

4: IMPROPER VARIABLE
- Reserved word being used or declared as a variable.
- Label not an identifier.

5: IMPROPER STATEMENT
- Statement does not contain the proper number of operators, i.e., a surplus of operands is present.
- Construct not a statement.

6: MISPLACED OPERATOR/MISSING VARIABLE
- More than one operator between operands.

7: MISSING OPERATOR
- No operator between operands.
- THEN missing in IF statement.

8: DUPLICATE NAME
- Multiple usage of the same label in a block, or redefinition of an identifier or formal parameter in a block or procedure heading.

9: TOO MANY BLOCKS
- Number of blocks plus procedures greater than 256.

10: MEMORY CAPACITY EXCEEDED
- Dictionary space exhausted.

11: MISSING BEGIN
- No matching BEGIN for an END.

12: MISSING LEFT PARENTHESIS
- No matching left parenthesis for a right parenthesis.

13: IMPROPER PARAMETER
- Formal parameter specified more than once.
- Formal parameter not specified in the specification part.
- Parameter in specification part not one of the procedure's formal parameters.
- Switch or procedure identifier in specification part also appeared in the VALUE part.

14: IMPROPER GO TO
- Value of designational expression is not a label.

15: IMPROPER SWITCH REFERENCE
- Switch designator does not contain one subscript expression.

16: IMPROPER ITERATION VARIABLE
- FOR statements control variable has BOOLEAN or STRING arithmetic.
- Control variable not declared.
- Control variable not a simple variable or array element.

17: MISSING THEN

18: MISSING ELSE

19: IMPROPER IF CLAUSE
- Expression between IF and THEN does not have BOOLEAN arithmetic.

20: MISMATCHED IF ARGUMENTS
- The quantities between the THEN - ELSE, and ELSE - END/; are not the same kind of expression, i.e., both are not arithmetic, boolean or designational expressions.

21: TEMPORARY STORAGE TABLE EXCEEDED
- See error 0.

22: MISPLACED COMMA

23: COMPILER CAPACITY EXCEEDED
- List space used by compiler has been exhausted.

24: IMPROPER CHARACTER
- EBCDIC character not defined in the language.

25: MISSING END
- No matching END for a BEGIN.

26: EXTRA LEFT PARENTHESIS

27: PARAMETER NOT SPECIFIED
- Using a formal parameter which was not specified in the procedure's specification part.

28: UNDEFINED ARITHMETIC OPERATION
- Arithmetic of operands not defined for the operator. For example, the operands for // are both not of type INTEGER.

29: INCOMPATIBLE ARGUMENTS IN AN EXPRESSION
- One of the operands of a binary operator cannot be converted to the arithmetic required by the operator.

30: IMPROPER TYPE CONVERSION
- Attempt to use a BOOLEAN expression arithmetically.
- Attempt to replace a BOOLEAN variable with an arithmetic expression.
- Attempt to replace a non-COMPLEX variable with a COMPLEX expression.

31: INTERNAL ERROR ARITHMETIC CONVERSION
- See error 0.

32: UNDEFINED ARITHMETIC CONVERSION
- See error 0.

33: MISSING RESULT
- Used a non-typed procedure in an arithmetic expression.

34: UNDEFINED VARIABLE
- See error 0.

35: INTERNAL ERROR OPERATOR VARIABLE
- See error 0.

36: IMPROPER LIBRARY PROCEDURE CALL
- Number of specified input arguments not equal to the number required.
- Attempt to pass the name of a non-recoursive library procedure as an argument to a procedure.

37: IMPROPER FOR STATEMENT
- Expression after WHILE not a BOOLEAN expression.
- Complex iteration variable on the STEP-UNTIL form.

38: ILL-FORMED PROGRAM
- Program not correctly structured.

39: VALUE SPECIFICATION OUT OF SEQUENCE
- VALUE part appears after specification part.

40: IMPROPER REPLACEMENT
- Left part not a simple variable, procedure identifier, partial string expression or partial bit expression.
- Left part is a procedure identifier not occurring in its declaration.
- Left part is a partial string operation on a string constant, result, or a non-variable (e.g., procedure identifier).
- Left part is a partial bit operation on a constant, result, or a non-variable.

41: IMPROPER FORMAT DECLARATION
- Improper syntax.
- Declaration does not occur in the block head.
- More in 132 characters in ' format.

42: MISSING ARRAY PARAMETERS
- Syntax of array list incorrect.

43: MISSING ARRAY BOUND
- Either the upper or lower bound of the bound pair is missing.

44: INCOMPLETE ARRAY DECLARATION
- Bound pair list not terminated.

45: IMPROPER ARRAY BOUND
- Array bound not an arithmetic expression.

46: VARIABLE NOT DECLARED
- Attempt to use a variable in an arithmetic expression that was not declared.

47: IMPROPER ARITHMETIC VARIABLE
- Variable in an arithmetic expression not a simple or subscripted variable, or a constant.

48: UNDEFINED OPTION SPECIFIED
- Option on !ALGOL card not permissible.

49: IMPROPER DESIGNATIONAL EXPRESSION
- SWITCH list element not a designational expression.

50: IMPROPER LOCAL DECLARATION
- Declaration does not occur in the block head.
- Word LOCAL not followed by the word LABEL, SWITCH or PROCEDURE.

51: RELOCATION COUNTERS EXHAUSTED
_ Program too large.

52: INCONSISTENT OPTIONS
- Both the FL and GO options specified on !ALGOL card.

53: IMPROPER EXTERNAL DECLARATION
- An external procedure not being declared.
- FL option specified on !ALGOL card.
- EXTERNAL not followed by INTEGER, REAL, BOOLEAN, ARRAY, STRING, COMPLEX, LABEL, PROCEDURE or METASYMBOL.

54: IMPROPER SPECIFICATION
- Specifier in procedure specification part preceded by the word LOCAL, OWN or EXTERNAL.

55: TOO MANY ERRORS
- Program contains 25 compile-time errors.

56: IMPROPER PARAMETER DELIMITER

57: DANGLING ELSE
- ELSE out of context.

58: IMPROPER IF EXPRESSION
- The quantity between the THEN - ELSE not a simple arithmetic, boolean, or designational expression.

59: VALUELESS TYPED PROCEDURE
- In the declaration of a typed procedure, the procedure identifier did not occur as the left part of an assignment statement.

60: IMPROPER COMPLEX OPERATION
- Use of an undefined COMPLEX relation.
- Use of a COMPLEX argument to a non-COMPLEX function.

61: MISPLACED COLON

62: PROCEDURE CALLS NESTED TOO DEEP
- Procedure calls have been nested more than 10 deep.

63: ABNORMAL I/O: RECORD LENGTH> 80
- An abnormal return other than end-of-data or end-of-file has occurred when reading from M:SI.

X1: MISPLACED PERIOD
- Bit operator not followed by bit designator.

X2: IMPROPER BIT DESIGNATOR
- Bit designator does not contain two arguments.

X3: IMPROPER STRING ARRAY DECLARATION
- Equivalence form of declaration being used in an OWN STRING ARRAY declaration.
- String array being set equivalent to a quantity other than another string array or a part thereof.
- Length of elements not specified.
- In equivalence form of declaration the sub-string designator is incomplete.

X4: IMPROPER BIT REPLACEMENT
- Bit operator applied to a procedure identifier occurring as the left part of an assignment statement.

X5: IMPROPER STRING DECLARATION
- Improper syntax.
- String being set equivalent to a quantity other than a string.
- Length of string not specified.

X6: IMPROPER PARTIAL STRING DESIGNATOR
- Syntax of partial string designator incorrect. Either '(' is missing or more than 2 operands supplied.
- Operand of partial string operator illegal.

X7: IMPROPER STRING CONSTANT
- String constant empty.
- Length of string constant greater than 255 characters.

X8: DEFINE IDENTIFIERS NESTED TOO DEEP
- DEFINE identifiers nested more than 8 deep.

X9: IMPROPER DEFINE DECLARATION
- Declaration does not occur in the block head.
- Improper syntax.

X10: IMPROPER BIT EXPRESSION
- Operand of bit operator illegal.

X11: IMPROPER DUMP STATEMENT
- DUMP list elements not separated by comma's, or the statement doesn't terminate with a ';'.

X12: ILLEGAL DUMP LIST ELEMENT
- DUMP list element is either not defined, local to the block or is other than a constant, simple variable, array name or value parameter.

X13: IMPROPER TYPE TRANSFER FUNCTION CALL
- Improper arithmetic for argument to a type transfer function.
- Number of specified input arguments not equal to 1.

II. RUN-TIME ERROR MESSAGES

1: OUT OF SEQUENCE 'OPEN'
- Attempt to do an OPENREAD/OPENWRITE again without first having done the corresponding CLOSEREAD/CLOSEWRITE.

2: IMPROPER 'OPEN' ARGUMENT
- Argument not a I/O Device number of format label.

3: 'OPEN' ARGUMENTS OUT OF SEQUENCE
- Arguments to OPENREAD/OPENWRITE reversed.

4: TOO MANY 'OPEN' ARGUMENTS
- 3 or more arguments specified to OPENREAD/OPENWRITE.

5: OUT OF SEQUENCE 'CLOSE'
- Attempt to do a CLOSEREAD/CLOSEWRITE again without first having done the corresponding OPENREAD/OPENWRITE.

6: IMPROPER 'CLOSE' CALL
- Arguments specified to CLOSEREAD/CLOSEWRITE.

7: OUT OF SEQUENCE I/O
- Attempt to do INPUT/OUTPUT without first having done an OPENREAD/OPENWRITE.

8: MISSING I/O LIST
- No I/O list specified for INPUT.

9: IMPROPER INPUT VARIABLE
- Item in I/O list not a variable, or array identifier.

10: IMPROPER OUTPUT VARIABLE
- Item in I/O list not a variable, number, array identifier or expression.

11: NO ARITHMETIC SPECIFIED FOR I/O VARIABLE
- Variable in I/O list not typed.

12: SUM CHECK ERROR
- Program cannot be loaded. See a Computer Center staff member.

13: RECORDS OUT OF SEQUENCE
- See error 12.

14: IMPROPER LOAD RECORD
- See error 12.

15: MISSING PROGRAM START
- See error 12.

16: MISMATCHED NUMBER OF SUBSCRIPTS
- Dimension of a subscripted variable does not agree with the dimension of its corresponding array declaration.

17: DROPPING TO UNDEFINED BLOCK
- Value of the designational expression in an EOF or ERR statement is not within the scope of the block containing the READ statement.

18: RAISING TO UNDEFINED BLOCK
- An error or end-of-file has occurred and a transfer is attempted to a ERR or EOF return point which was set in a block that no longer exists.

19: RAISING TO UNDEFINED BLOCK FOR NOP

20: NO ARGUMENTS IN READ/WRITE CALL

21: REGISTER STACK OVERFLOW
- Too many levels of recursion.

22: IMPROPER RECURSIVE FUNCTION CALL
- A methematical function name passed as a parameter to a procedure has been used with an incorrect number of arguments, or the argument is not valid for the function.

23: SUBSCRIPT TOO SMALL
- Subscript in a subscripted variable is less than the lower bound specified in its corresponding array declaration.

24: SUBSCRIPT TOO LARGE
- Same as 23, except subscript greater than upper bound.

25: STORAGE ALLOCATION ERROR
- The program has requested more space than is available.
- The program tried to return more space than it had.

26: IMPOSSIBLE PARAMETER
- Actual parameter not permissible. This error should be considered an internal run-time error. See a staff member of the Computer Center.

27: MISMATCHED NUMBER OF ARGS TO PROC
- Number of actual parameters supplied in a procedure call not the same as the number of formal parameters specified in the procedure declaration.

28: MISMATCHED TYPES
- The kind of quantity supplied in a procedure call does not agree with the corresponding formal parameter.

29: ARITHMETICS MISMATCHED FOR NAME ARG
- A formal parameters that is called by name does not have the same type as the actual parameter.

30: ENTIER ARG TOO BIG OR TOO SMALL
- Floating point number can't be converted to integer.

31: UNDEFINED ARITHMETIC FOR VALUE ARG

32: INVALID VALUE ARGUMENT
- Actual parameter that corresponds to a formal parameter specified as call by value is not a variable, number, expression, string, statement label, format, or array.

33: LENGTH OF CONCATENATION > 255

34: TOO MANY ARRAY DECLARATIONS
- Number of array identifiers associated with a bound pair list in the declaration is greater than 15.

35: START POSITION OR LENGTH <1

36: UPPER BOUND < LOWER BOUND

37: INVALID PROCEDURE PARAMETER
- The actual parameter is a procedure call, instead of a procedure identifier.

38: IMPOSSIBLE ARITHMETIC CONVERSION
- Cannot convert the actual parameter in a procedure call to the type of the corresponding formal parameter.

39: IMPROPER BIT ACCESS
- Starting value or number of bits referenced <1.

40: REFERENCING TOO MANY BITS
- Bit length exceeds data structure size.

41: STARTING BIT VALUE TOO LARGE

42: IMPROPER BIT REPLACEMENT
- Starting value or number of bits reference <1 in replacement.
- Trying to store bits outside the boundaries of a data structure.

43: TRYING TO STORE MORE THAN 32 BITS

44: MISMATCHED ARITHMETIC ON STRING ARRAY PARAMETER
- String arrays may only be passed into string arrays as value parameters.

45: STRING LENGTH < 1
- Occurs for non-array string declarations.

46: TRYING TO ACCESS TOO MANY CHARACTERS

47: INVALID STRING USED AS AN INTEGER
- See reference manual for proper form of integer string.

48: SUB-STRING LENGTH LONGER THAN ARRAY'S
- The length specification for each element of a substring array exceeds the length specified for each element in the original array.

49: LENGTH OF STRING > 255
- Occurs for string declarations.

50: IMPROPER STRING ACCESS/REPLACEMENT
- Number of characters referenced < 1.
- Index of the initial character < 1.
- Index of the initial character)length of the string.

51: ARITHMETIC FAULT
- Division by zero.
- Arithmetic overflow or underflow (number too large or small)

NOTE: To facilitate debugging, the contents of the registers, and the instruction being executed and its location are also printed. Furthermore, use of the DM option causes the line number of the statement creating the error to be printed.

52: ILLEGAL 'EOF' or 'ERR' CALL
- Argument not a designational expression, i.e., not a label.
- More than one argument.
- Value of the designational expression is undefined, i.e., the designational expression is a switch designator whose value is undefined.

53: INVALID ARGUMENT(S) TO FUNCTION XXXX
- The FORTRAN run-time math routine XXXX detected an invalid argument.

54: PROGRAM ABORTED
- Number of compile-time errors greater than 15.
- Number of compile-time errors greater than the severity level specified on the !ALOAD card.
- ALOAD misspelled.